Early Address Prediction: Efficient Pipeline Prefetch and Reuse

RICARDO ALVES, STEFANOS KAXIRAS, and DAVID BLACK-SCHAFFER,
Uppsala University

Achieving low load-to-use latency with low energy and storage overheads is critical for performance. Existing techniques either prefetch into the pipeline (via address prediction and validation) or provide data reuse in the pipeline (via register sharing or L0 caches). These techniques provide a range of tradeoffs between latency, reuse, and overhead.

In this work, we present a pipeline prefetching technique that achieves state-of-the-art performance and data reuse without additional data storage, data movement, or validation overheads by adding address tags to the register file. Our addition of register file tags allows us to forward (reuse) load data from the register file with no additional data movement, keep the data alive in the register file beyond the instruction's lifetime to increase temporal reuse, and coalesce prefetch requests to achieve spatial reuse. Further, we show that we can use the existing memory order violation detection hardware to validate prefetches and data forwards without additional overhead.

Our design achieves the performance of existing pipeline prefetching while also forwarding 32% of the loads from the register file (compared to 15% in state-of-the-art register sharing), delivering a 16% reduction in L1 dynamic energy (1.6% total processor energy), with an area overhead of less than 0.5%.

CCS Concepts: • **Computer systems organization → Superscalar architectures**; **Pipeline computing;**

Additional Key Words and Phrases: Pipeline prefetching, first level cache, energy efficient computing, address prediction, register sharing

ACM Reference format:
Ricardo Alves, Stefanos Kaxiras, and David Black-Schaffer. 2021. Early Address Prediction: Efficient Pipeline Prefetch and Reuse. *ACM Trans. Arch. Code Optim.* 18, 3, Article 39 (June 2021), 22 pages.
https://doi.org/10.1145/3458883

1 INTRODUCTION

Satisfying loads as early as possible is critical for performance. As it is difficult to build faster L1 caches, many approaches have been proposed to reduce load latency by prefetching data directly into the pipeline [5, 39] just in time for consumption to achieve zero load-to-use latency. The downside of pipeline prefetching is that the prefetched loads, because they used predicted addresses,

New article, not an extension of a conference paper.
Authors' addresses: R. Alves, 2111 N.E. 25th Avenue, Hillsboro, OR 97124, United States of America; email: ricardo.alves@intel.com; S. Kaxiras and D. Black-Schaffer, Uppsala University, Box 337, 75195 Uppsala, Sweden; emails: {stefanos.kaxiras, david.black-schaffer}@it.uu.se.

1544-3566/2021/06-ART39 $15.00
https://doi.org/10.1145/3458883

require validation. This means that pipeline prefetches need to be re-executed: re-loading their value from the memory system and verifying that the value used speculatively was correct. This is necessary to account for local and remote stores and results in a doubling of L1 cache accesses for all pipeline prefetches.

This problem can be addressed by installing prefetched data in an intermediate storage between the CPU and the L1 [15, 27]. However, this strategy requires extra storage area and data movement to an intermediate location between L1 and the register file, which also has to be exposed to coherence traffic for correctness. Energy-wise, this approach is similar to filter-cache/L0 solutions [6, 14, 18], which also pay the energy overheads of moving and installing data in extra storage despite achieving only limited locality [2].

A ubiquitous trait of these pipeline prefetching techniques is that the address predictors required to prefetch loads make the addresses of load instructions available in the front-end of the CPU. The early availability of addresses has significant implications for the way data can be tracked in the **physical register file (PRF)**. Since the PRF is the final destination of loads, it can be seen as truly the lowest level of the cache hierarchy. However, load-to-load data reuse (typical of all other cache levels) is difficult to expose through the PRF due to the tracking of data in being done in the *register-space* (used by instructions) and not in the address-space (as used in the cache hierarchy). The reuse potential of the PRF has been explored before [28, 32], but previous work has had to rely on heuristics to track reuse, precisely because they did not have addresses early in the pipeline, which significantly limited its potential.

Our main insight is that *having address predictions early in the pipeline allows us to detect load-to-load reuse before the rename stage where registers are allocated.* This allows us to *reuse registers at rename*, instead of having to re-fetch data from memory and re-install the same data in another register. While previous pipeline prefetching techniques either increase pressure on the L1 [5, 39] or require extra data storage [15, 27], our solution not only avoids both problems, but actually reduces L1 accesses by forwarding the data from the PRF when reuse is detected. Moreover, our solution reduces capacity pressure in the physical register file beyond traditional register-sharing techniques due to the increased reuse detection of our address-based approach, while preserving the performance benefits of existing pipeline prefetching techniques. While our solution comes with additional memory ordering problems, we show that these can be solved by re-purposing existing pipeline structures such as the load queue (to solve coherence violations) and the memory order violation detection hardware [12, 37] (for validating both data prefetches and load-to-load forwarding through the PRF).

In summary, our contributions are:

- We identify that having an address prediction for loads early in the pipeline enables us to effectively combine pipeline prefetching, register reuse, load coalescing, and cheap validation.
- We demonstrate that adding address tags to the PRF allows us to find reuse via address prediction at rename and keep data in the PRF after the instruction completes. As a result, we can achieve long-term temporal reuse, but without the storage and data movement overhead of an L0. (We only add metadata storage to map addresses to registers.)
- We take advantage of having address tags for the PRF to store prefetched values. This achieves the temporal reuse of dedicated prefetch storage, but with the same zero-overhead data storage of just-in-time prefetching.
- We take advantage of the cache bus returning more than one word to coalesce neighboring prefetches and install them into the PRF. This provides spatial reuse, but without the data storage and data movement overheads of an L0.

- We demonstrate that the existing memory order violation predictor can be extended to validate both pipeline prefetches and data forwarding from the PRF, eliminating the need for dedicated structures or re-issuing the loads.

Our solution is able to reuse (forward) 32% of the loads from the register file, compared to only 15% for a state-of-the-art register-sharing technique [32], thereby capturing 93% of the available load-to-load reuse potential in the instruction window. Compared to just-in-time pipeline prefetching [39], our solution avoids replaying over 99% of the instructions for validation, without the overhead of extra validation mechanisms nor the dedicated data storage of previous approaches [15, 27]. As a result, we reduce the total L1 load accesses by 32%, while delivering the performance benefits of existing state-of-the-art pipeline prefetching. By using the PRF for data storage, we are able to accomplish this with only the overhead of a small tag array (0.5% of the CPU core area).

2 BACKGROUND

2.1 Pipeline Prefetching and Value-prediction

Value prediction (VP) [21, 22, 29, 31, 41, 42] guesses the result of an instruction before its execution. This allows instructions to proceed with predicted operands without waiting for the instructions that generate these operands to execute. This *speculatively* breaks true data dependencies and thus has the potential to increase **instruction level parallelism (ILP)** beyond the data flow limit. VP comes with the caveat that mispredictions are costly, so predicted values are only used when the prediction has a high confidence, leading to low coverage. Furthermore, since stores change values in memory, they often interfere with predictions, further decreasing confidence (and coverage) [39].

Value prediction techniques can be divided into two classes: *computation-based* (such as stride predictors [10, 13]) and *context-based* (such as VTAGE [30] and DVTAGE [31]). Orosa et al. [27] recently proposed an hybrid that merges a stride predictor with DVTAGE to improve on both.

Sheikh et al. [39] made the observation that addresses are easier to predict than data, and proposed the ***Decoupled Load Value Prediction* (DLVP)**. DLVP predicts load *addresses* instead of the load values and issues the loads as a *pipeline prefetchs* just-in-time to be consumed by the dependent instruction(s). Predicting addresses has two main advantages: First, addresses have simpler patterns, thereby increasing prediction coverage, reducing iterations required to trust the prediction, and generating fewer mispredictions due to stores changing the predicted value. And, second, since approximately 25% of instructions are loads, the predictor does not need to store nearly as much state [27], making the prediction mechanism smaller and less complex. Moreover, recent work on pipeline prefetchers has shown that predicting the address of loads and prefetching them from memory achieves the same performance benefit as predicting all values for all instructions [27, 39].

However, while effective, just-in-time pipeline prefetching significantly increases memory accesses, as every pipeline prefetch needs to re-access memory a second time to validate the result, in case the prediction was wrong or there was a coherence or memory ordering violation. To avoid the overhead of re-accessing memory for validation and take advantage of potential data locality, the prefetched data can be installed in an intermediate storage between the pipeline and the first level of cache [27]. González et al. [15] further mitigate the secondary access for validation by leveraging the ***Address Reordering Buffer* (ARB)** [12] to filter replays. However, these solutions require installing data in a extra storage level, probing this extra storage level on every access, and exposing it to coherence. As a result, these intermediate storage approaches come with the same storage and energy overheads of filter-caches/L0s, which are generally unable to reduce total memory accesses [2].

2.2 Register Sharing

Register sharing was originally proposed by Jourdan et al. [16] to eliminate register moves. Their mechanism detects register reuse between instructions, renames the reused registers as the source registers of the new instructions, and tracks the lifetime of the shared registers.

Several techniques have been proposed to improve the efficiency of register tracking and sharing [4, 11, 28, 34, 35, 38]. Besides reducing register pressure, these technique also allow for zero load-to-use latency on loads that have their data forwarded by the physical register file. Most recently, Perais et al. [32] proposed the **Inflight Shared Register Buffer (ISRB)** to both improve reuse detection and simplify register tracking, as well as enable load-to-load data forwarding through the PRF. While such load-to-load forwarding is similar to our proposal, it requires cache accesses to validate the values of data forwarded to loads, has limited coverage due to the use of an heuristic for detecting reuse (instruction distance), and can only take advantage of temporal reuse.

2.3 Detecting Memory Ordering Violations

Memory ordering violations can come in two forms: *internal*, where a load incorrectly bypasses an older store on which the load depends, and *external*, due to memory consistency model violations, typically appearing as coherence invalidation requests that catch speculatively performed loads out of their correct memory order.

Internal memory violations happen because processors allow loads to bypass independent stores for performance, i.e., ***Speculative Memory Bypassing* (SMB)** [8, 17, 24]. However, this bypassing happens before addresses are available, and is therefore speculative, based on a prediction of the load's dependence on older stores. If the prediction is incorrect, then the load and its dependency chain (or simply all following instructions) need to be replayed. External memory ordering violations can occur due to speculatively reordered accesses that conflict with external accesses. This is handled by tracking incoming coherence events.

Ordering violations can be solved by a *value-based approach*, which does a brute-force re-execution of every speculative load at commit to compare the speculatively loaded value with the actual value. This comes with the cost of extra memory/cache accesses for each speculative load, which can have a significant impact on performance and energy. To address this problem, strategies have been proposed to validate most memory bypassing and coherence events while minimizing the replaying of loads [26, 34, 40]. For detecting internal violations, the most successful strategy uses a ***Store Sequence Bloom Filter* (SSBF)** [37] to avoid over 99% of replays.

External memory violations are detected by exposing the **load queue (LQ)** to coherence traffic. The load queue holds the physical addresses of each load after translation and generates replays on invalidation requests hits. Virtual addresses are also stored in the load queue to facilitate store-to-load forwarding and detection of internal memory ordering violations, since they are available earlier than physical addresses, using them can reduce the penalty of mispredictions.

3 MOTIVATION

(Figure 1(A)) In a baseline out-of-order processor, loads generate their addresses and then access the cache. Once the cache returns data, dependent instructions can execute. However, as loads and stores may bypass each other during execution, a final validation that the load's value is correct with regards to the *internal ordering* of instructions is required at commit. External ordering violations are caught by the LQ as it participates in coherence. In this work, we assume the internal ordering commit validation is done using SSBF [37], which can validate over 99% of all accesses without accessing the L1.

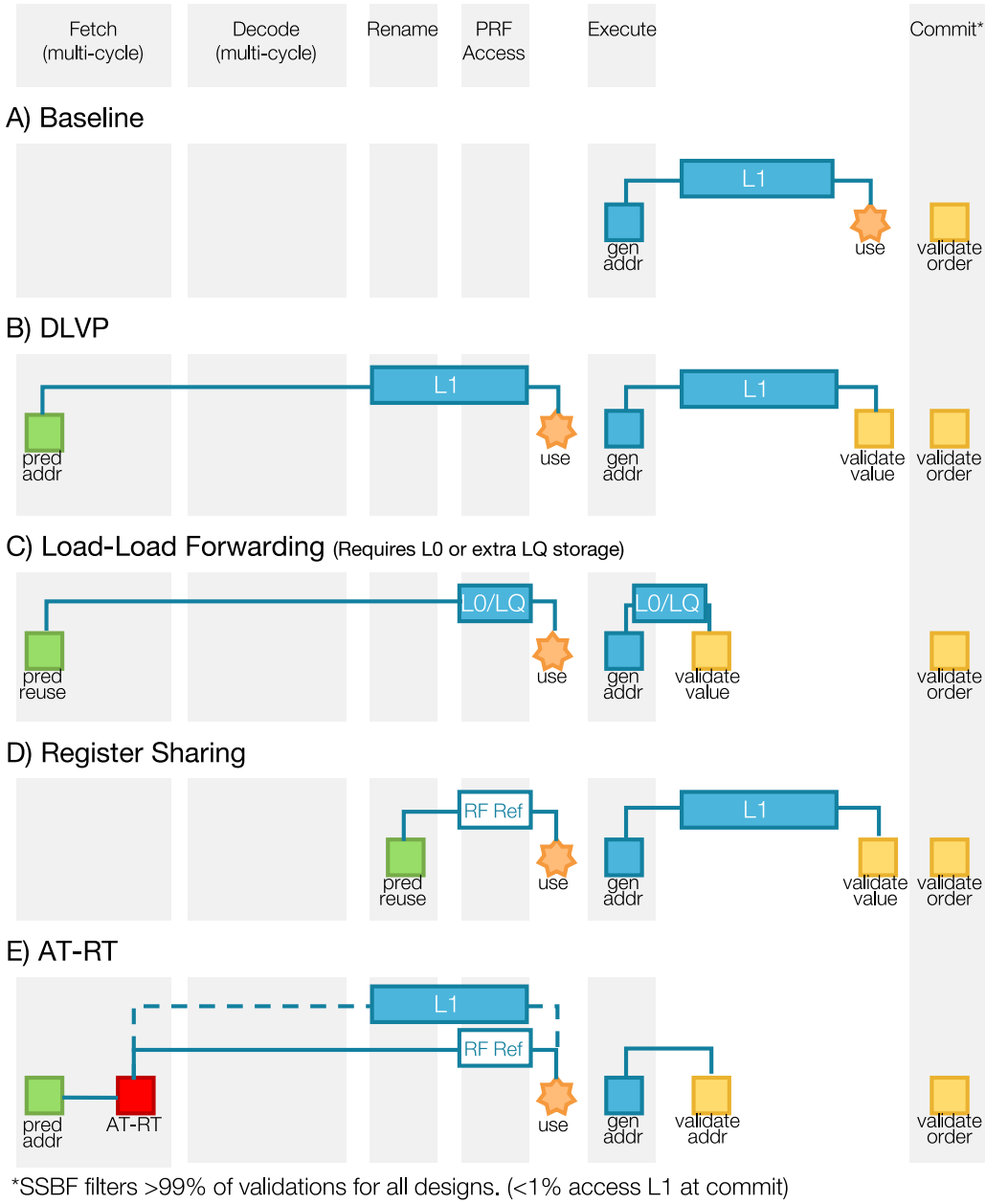

Fig. 1. Comparison of when data is available (*use*) and structures accessed. Register Sharing and AT-RT need only a *reference* to the register (not the data). Note that the pipeline stages are approximate to show the relative timings.

(Figure 1(B)) *DLVP's* [39] just-in-time pipeline prefetching predicts the *load addresses* and accesses the L1 earlier in the pipeline. This allows the cache to deliver the data by the time the dependent instructions are ready and achieve 60% coverage (Figure 2). However, DLVP requires a second L1 access once the load address is generated to validate the value, as well as the final ordering validation at commit.

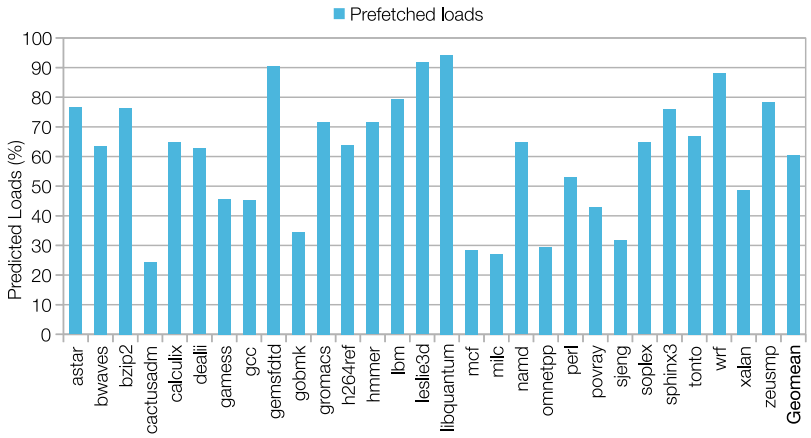

Fig. 2. Ratio of loads that a stride address load predictor covers and that DLVP would prefetch.

(Figure 1(C)) *Load-to-load forwarding* exposes locality by adding extra storage (an L0 or an extended LQ) to keep copies of the loaded data closer to the processor [15, 27]. This allows later instructions to get the expected value of the load from the storage earlier in the pipeline. When the load's address is later generated, the same storage structure can be used to validate that the correct value was loaded, as the storage is searchable by the final generated address. This minimizes pressure on L1, but requires extra storage and extra data copies. The second access to validate internal ordering can be avoided [15] but at least one access to the L1 or intermediate storage is needed for every load.

(Figure 1(D)) *Register Sharing* can eliminate the need for additional data storage by using the physical register file. This also, and perhaps more importantly, avoids the need to access the cache and copy the data when a register reuse is predicted as the rename logic simply sets the appropriate register reference for the load. This achieves the same zero load-to-use latency of pipeline prefetching without extra storage or data movement. However, when the load's address is later generated, an L1 access is still required to validate internal memory ordering and coherence, since the register file is not exposed to coherence traffic. However, register-sharing is unable to provide high coverage due to its reuse heuristics [32] (instruction distance in Figure 3), which can only exploit less than half of the load-to-load forwarding potential that exists in the pipeline at any given time.

State of the Art summary: While DLVP's pipeline prefetching 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 has the potential to mitigate the increased L1 pressure by satisfying prefetch requests from the register file, but previously proposed approaches have shown limited reuse detection. Our proposal, AT-RT, addresses these challenges while delivering the same zero load-to-use latency as DLVP.

(Figure 1(E)) Our proposal (AT-RT) takes advantage of DLVP's early address prediction and extends it to *identify reuse in the register file* without the need for L1 accesses or data copies. If pipeline prefetches are found in the register file, then we do not require a memory access, resulting in reduced L1 pressure and energy. AT-RT accomplishes this by providing a table to index the register file by address (i.e., register file tags). By searching for register reuse by address, AT-RT is able to dramatically increase reuse detection over previous techniques, thereby reducing register file pressure, and take advantage of both spatial and temporal reuse. Further, as the reuse prediction

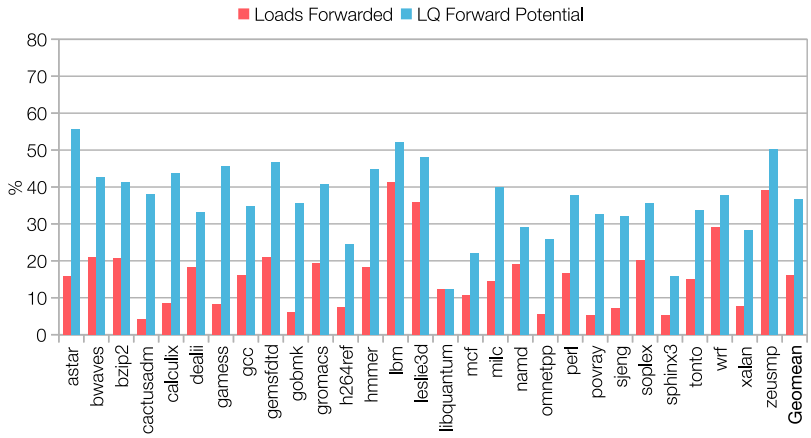

Fig. 3. Ratio of loads that are forwarded using instruction distance to detect reuse, compared to total reuse potential available in the LQ.

Table 1. Comparison of the Techniques Considered in this Work

Strategies	Prefetch	Granularity	To Where?	Load-Load reuse	Validation	L1 accesses	Registers used
DLVP [39]	✓	accessed item (e.g., word)	pipeline	✗	Value-based	more	same
AVPP [27]	✓	next line (e.g., next word)	scratchpad (extra storage in LQ)	✓(temporal & spatial)	Value-based	fewer	same
ISRB [32]	✗	–	–	✓(temporal only)	Value-based	same	fewer
AT-RT	✓	block (e.g., 128 bits)	PRF	✓(temporal & spatial)	Address-based (99%) Value-based (1%)	fewer	fewer

is made based on the *address*, AT-RT can then use the predicted address to validate the load against its final generated address, avoiding the need for a second access. If the addresses match, then only a final internal ordering validation is needed. A comparison to related work is provided in Table 1.

4 EFFICIENT TEMPORAL LOAD TO LOAD FORWARDING AND PREFETCHING

The ability to accurately predict the address of a load from its PC in the fetch stage allows us to speculatively fetch the value from the L1 without having to wait for the load address to be computed. However, to be able to take advantage of reuses of this load in the future, it must be cached close to the pipeline. Using the physical register file is a compelling choice, as it is already the final destination for the data, which means there is no additional data movement overhead.

To effectively use the PRF as a cache, we need to address two issues: (1) *finding* the prefetched data and (2) *forwarding* it without having to re-access the L1. We achieve this by providing an address-to-register mapping (essentially register file tags) that allow us to find load data in the PRF based on its address. By making the PRF work as an address-based cache, we are able to overcome the coverage limitations of previous work (since we can find all address reuses and are

not limited to inter-instruction heuristics) and avoid extra cache accesses for validation (since we can use the final address to validate execution order).

4.1 Finding and Forwarding Prefetched Data via the Address Tag-Register Tag (AT-RT) Table

To detect register reuse and perform load-to-load forwarding through the physical register file, we introduce a register translation table, the ***Address Tag, Register Tag* (AT-RT)** table. The AT-RT provides address tags for the entries in the PRF that were loaded from memory. The direct-mapped AT-RT contains a register tag, the data size, reference counter for the number of load instructions that are sharing the register, a valid bit, and an address tag for each entry.

Load instructions put their destination register tag and (predicted) address tag in the appropriate AT-RT entry, increment its reference counter, and mark it as valid. The table is then indexed using the predicted address for each later load in the fetch stage. When a later load finds its predicted address in the AT-RT, it skips allocating its own destination register and instead reuses the same register tag as the one found in the AT-RT by simply incrementing its reference counter. This allows loads to treat the PRF as a cache by detecting and reusing registers based on the address they correspond to. Note that all of the above is enabled by having a predicted load address before register renaming in the pipeline.

The AT-RT reference counters are independent from existing reference counters in the PRF. Thus, freeing an entry in the AT-RT does not automatically free the corresponding physical register. However, as we will show, AT-RT entries can serve as hints to prevent register de-allocation for data that may see reuses and thereby increase reuse.

Prefetched loads are inserted in the load queue with their predicted address to enable address-based validation later on without extra storage or tracking mechanisms. After the load reaches the execution stage, its effective address is generated and compared to its predicted address. If the addresses match, then the prediction is deemed accurate and the load does not have to be replayed for validation. (The detection of possible memory order violations may cause a replay as with any out-of-order pipeline; see Section 4.2).

4.2 Forwarding Validation

Validating forwarded data requires ensuring that the address prediction was correct and that there were no internal or external ordering violations between the time the forwarded data was used and when the instruction committed in program order. The address predictions can be easily validated when the load's address is finally generated by simply comparing the generated address to the predicted address in the LQ. Note that the pipeline prefetcher predicts in the virtual address space, so prefetches will still need to be translated, and we can reuse the translation if the predicted and final virtual addresses match. The baseline (Figure 1(A)) requires the same translation, but it happens later, as there is no pipeline prefetching. Validating memory order is more involved and can be broken down into two separate problems, external and internal violations, discussed below.

External Violations. External violations come as memory coherence invalidation requests. Although complex, this type of ordering violations is present in all multi-core systems and the CPU already has mechanisms to detect the violating loads [37], rollback the values of the shared registers [32], and replay the infringing instruction chains [33]. The only extra step introduced by our approach is that the AT-RT also needs to be accessed to invalidate any entries for the invalidated addresses to avoid forwarding stale data. While the AT-RT table is virtually tagged, the LQ, which holds both virtual and physical addresses for responding to coherence snoops, can provide translations to find the correct entry. If the LQ does not have a physical address translation yet, then it means that that particular load was not yet issued so it can not be in violation of the

consistency model. Since the AT-RT forwarding detection is done in the front end of the CPU, it is done in-order, and, as such, forwarding is always performed from older to younger loads. The invariant in our design is that a snoop always catches the *oldest* infracting load with invalid forwarded data.

Note that simultaneous accesses to the AT-RT table (simultaneous renaming and coherence) would not stall the pipeline, as loads can correctly be renamed and dispatched without consulting the AT-RT for reuse opportunities. Recent work has also shown that delaying coherence requests in the L1 could avoid the need to invalidate entries on invalidation requests [36].

Internal Violations. Internal memory order violations occur when loads incorrectly bypass dependent stores. This is an undesired effect of speculative memory bypassing, a common technique to improve performance on out-of-order pipelines. All pipelines allowing speculative memory bypassing need to detect such violations. While our approach is independent of the mechanism chosen, we implemented the SSBF proposed by A. Roth [37], as it minimizes redundant memory accesses (<1%) to the L1. SSBF works by filtering loads that are not in a ***Store Vulnerability Window* (SVW)**. This 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.

Figure 4 shows an example of how order is validated by an SSBF as proposed in the original work. At commit, and in program order, the stores update the SSBF table and loads check the SseqN of the youngest store that wrote to the same address. If the youngest store to write to that address is older than the store the load bypassed, then the issue order is valid, otherwise, there may have been an order violation and the load needs to be replayed. The example shows an execution order that has the first load correctly bypassing an older store and a second load correctly bypassing one store (ST_A) but incorrectly bypassing a second (ST_B). At commit, the infracting load is correctly detected, since it bypassed an older store than the last one that wrote to same memory location.

While SSBF cannot be paired with register reuse or pipeline prefetching techniques as originally proposed, it can be extended to do so. For pipeline prefetched data, the validation needs to be done by comparing prefetch order (instead of execution order) with program order as in the original design (as exemplified in Figure 4). This is a simple modification and similar to the one proposed by González et al. [15]. However, this modification alone is not enough to validate load-to-load forwarding through the register file. While load-to-load forwarding does not affect the scheduling and execution order of the associated loads, by forwarding the data from older loads to younger ones there is an implicit memory bypassing by the younger load. From a memory ordering point of view, both loads "executed" at the same time. This can be an issue if there is a conflicting store between two loads with forwarding. To address this problem, we modify SSBF to have loads inherit the SseqN of the loads they had the data forwarded from, and thus inherit their vulnerability window as well. This modification enables the memory ordering validation strategy (in this case SSBF) to function properly and avoid accessing the memory for validation on both load-to-load forwarding and register-sharing techniques.

4.3 Interaction with the Memory Bypass Predictor

Any speculative forwarding technique needs to be informed of memory dependencies by the speculative memory bypassing predictor, as misschedulings due to missed dependencies can force pipeline flushes and have a profound impact on performance [1, 33]. We use the same Load Store Conflict Detector [39] strategy as DLVP to enable or disable forwarding for particular loads before rename. If the LSCD detects a forwarding conflict, then we simply invalidate the corresponding entry in the AT-RT table, preventing further forwarding from loads that are likely to have stale

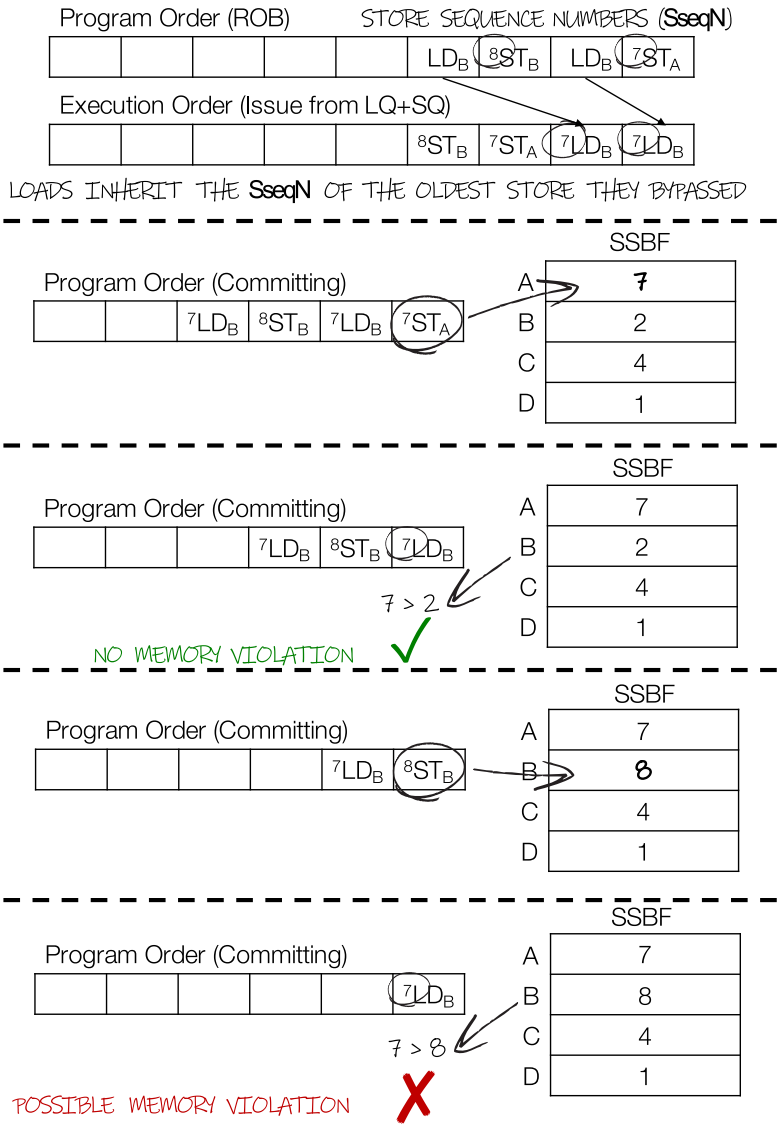

Fig. 4. Example of how the Store Sequence Bloom Filter (SSBF) [37] is used to detect possible memory ordering violations using Load/Store addresses (letters) and store-sequence-numbers (numbers).

data. Figure 5 shows an example of a typical memory ordering violation introduced by load-to-load forwarding and the correct forwarding combination.

Perais at al. [32] went one step further to break memory dependencies by identifying the producer of a load and renaming the destination register of the load as the source register of the producing store. This effectively allows for load-to-store forwarding through the register file as well. In this work, we consider only load-to-load forwarding, because, unlike loads, stores already have dedicated storage in the pipeline for their data: the store-queue and the store-buffer. Data produced by stores must be installed in this storage regardless, and every load has to probe it for correctness [36], so there is no energy/storage benefit in forwarding the data from the PRF

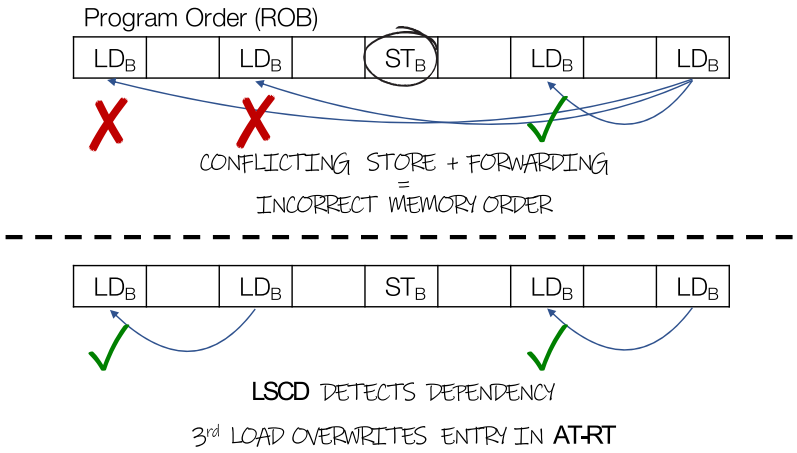

Fig. 5. Example of how forwarding can create a memory ordering violation even if execution follows program order. LSCD [39] learns the memory violation and breaks the erroneous load-load forwarding to the third and fourth loads by forcing the third load to make a new entry in the AT-RT. This also enables the correct forwarding to the fourth load.

directly. The store-buffer can be used to filter accesses to L1 [3], but the access to the store-buffer is still required, even if data is to be forwarded from the PRF. Note that here is no overlap between load-to-load and store-to-load forwardings.

4.4 Address Prediction

Our solution is independent of the chosen address predictor but needs the prediction early in the pipeline to manage register sharing and provide load-to-load forwarding through the PRF. For simplicity, we choose a stride-based predictor. The prediction table is indexed by PC and each entry holds the last accessed address, the stride, and the confidence level (2-bit saturating counter; details in Section 6.1). The prediction is only trusted when the counter is saturated and our simulation results showed an average error ratio of only 1.6% (Section 6.2).

Although highly accurate, a stride predictor is not as accurate as a hybrid DVTAGE + Stride address predictor [27]. However, we found that the stride predictor was accurate enough to deliver significant benefits and provides a good baseline. *More sophisticated address predictors would simply increase coverage and/or decrease errors, improving our proposed solution even further, as our reuse detection and forwarding technique is independent of the type of address predictor used.*

5 EXPOSING SPATIAL LOCALITY VIA THE PRF

A key function of caches (indeed, the main benefit of L0s) is that they expose spatial locality. However, for pipeline prefetching, as long as the prefetch is on-time, there is no performance benefit to providing spatial locality. Moreover, pipeline prefetching minimizes pollution, as it only fetches the data that it predicts will be used, instead of fetching and installing adjacent data.

While the performance benefit of spatial locality is likely to be covered by prefetching, the energy benefit is not. This is because CPUs have wide L1 data busses (up to 512 bits) to satisfy vector loads, meaning that L1 loads already pay the energy cost of moving adjacent words. As a result, if we can take advantage of spatial locality, then we can move the data at no additional cost.

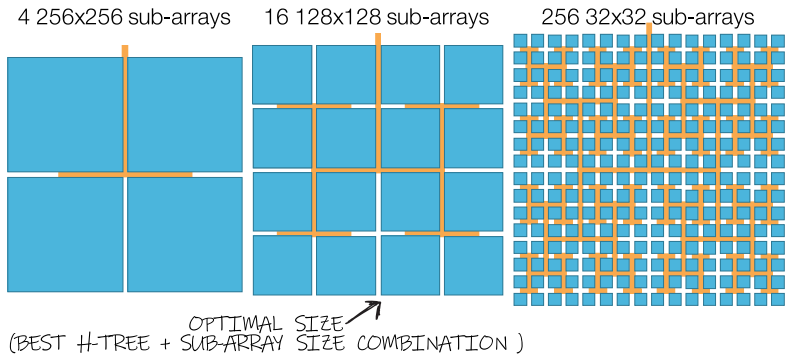

Fig. 6. 32 KB cache with using different sub-array size and respective h-tree combinations [2]. The smaller the sub-array, the smaller the minimum data read-out width, but the larger the overhead of the h-tree.

5.1 SRAM Cache Layout

SRAMs are built as matrices of smaller sub-arrays and connected with h-tree busses. 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. These sub-arrays can be resized, but smaller sub-arrays (which have smaller minimum read-out widths) require more sub-arrays for the same capacity, and thus larger h-trees. Since decreasing the sub-array size means increasing the h-tree complexity, the final design is usually a compromise (Figure 6). For a first-level, 32 KB, 8-way set associative cache, a 128-bit sub-array is a typical choice [2, 20], and we assume such a layout for this work.

As the sub-array determines the SRAM read-out width independently of the load instruction, all read-outs pay the energy cost of accessing and moving 128-bits. This means that energy benefit of coalescing multiple loads into 128-bits can be substantial. For example, grouping two 64-bit requests into a single 128-bit one (if they access the same sub-array line) could avoid a second 128-bit read, and thereby halve the cache access energy.

5.2 Coalescing Loads Efficiently

While storing all 128-bits from a load in the PRF is appealing to minimize L1 accesses, there are two main challenges: (1) increased writes and capacity pressure on the PRF from installing extra data that may not be used and (2) finding the speculatively installed data for use by later loads. Both of these problems can be solved by leveraging predicted load addresses: (1) Since loads' predicted addresses are available in the front-end of the CPU, they can be used to detect coalescing opportunities while the data is in-flight back from the L1 and only install the data from the 128-bit block that is predicted to be used. Given the small prefetch range (128 bits), the time between rename (where possible coalescing is detected) and load write-back (when we have to decide which parts of the 128-bit block to write in the PRF) is enough to find most loads that map to the same 128-bit block. (2) Since we only prefetch data for loads at or beyond the rename stage, all loads will already have a destination register in the AT-RT table. As a result, the 128-bit block that returns from the cache can be stored across the registers already mapped by the AT-RT that match the sub-addresses of the block. The bits of the 128-bit block that do not have a valid mapping in the AT-RT by the time the load returns, are simply discarded, i.e., only the data from the 128-bit block that was predicted to be used will be installed in the PRF.

This motivates our coalescing strategy: While we are waiting for a prefetch to return from the L1, we look for other prefetches that are predicted to access the remainder of the 128-bit block that the first prefetch will bring in. When the first prefetch returns, we install data into the PRF from

the 128-bit block for both the original prefetch and any other prefetches that were predicted to be in the block.

To optimize for this type of coalesced access, the AT-RT table can be modified so each entry tracks the group of registers that contain parts of the 128-bit block. Each entry then tracks multiple registers depending on the load size, i.e., 8, 4, 2, or 1 registers for load sizes of 1, 4, 8, or 16 bytes, respectively. This mapping does not change the behavior of tracking individual registers, since reference counting in the register file is done independently of how many registers each individual AT-RT table entry tracks.

5.3 Exposing more Locality than the LQ Window

Registers in the PRF are typically freed when their consumers commit. However, by freeing registers when their consumer commits, the forwarding potential of the register is limited to the processor's execution window, and, in particular, the number of loads that can be tracked at a given time. The reuse potential for loads is therefore capped by the reuse windows exposed by the load queue. This limitation also exists in previous register reuse techniques based on instruction distance, as the processor's instruction window determines the largest reuse distance that can be observed.

However, since AT-RT uses addresses to detect reuse, the lifetime of the load instruction no longer determines the time span across which we can detect reuse: Even if a load retires and leaves the load queue, it can still leave a valid address entry in the AT-RT table and data in the PRF. This means that another later load can be predicted to have the same address and reuse the data from the PRF. As a result, there is a tradeoff between freeing registers early to reduce PRF pressure and keeping them alive longer to increase PRF reuse. However, as register reuse inherently (and significantly) reduces PRF pressure, it makes sense to bias the tradeoff towards finding more reuse.

To take advantage of the increased reuse potential available due to using addresses to track register reuse, we have each entry in the AT-RT table increment the PRF reference counter as well. This way, even if all instruction referencing the register (loads or otherwise) commit, the PRF will not free the register if there is still a valid entry in the AT-RT that can allow later loads to reuse its data. Invalidating the entry (either to avoid memory ordering violations or a collision) requires decrementing the respective PRF reference counter as well.

With this strategy alone, AT-RT entries can only be invalidated if another address maps to the same entry (collision in the AT-RT table) or if a load-store collision is detected (LSCD unit). To allow the front-end of the CPU to free registers in case of increased register pressure, we need to implement register garbage collection. This system can be as simple as randomly invalidating AT-RT entries when needed to a full-blown LRU policy. Since we opted to track multiple registers per AT-RT entry to take advantage of the SRAM read-out width, each AT-RT entry has its own reference counter for how many loads in the pipeline are referencing it. As a result, the AT-RT entry is kept alive (and all its tracked registers) as long as at least one of its registers is referenced by an uncommitted load. This strategy does not maximize reuse, since we may still free registers too soon and miss out on further reuses. However, it strikes a compromise between extending a register's lifetime and implementing a trivial mechanism to recover unreferenced registers. Since the AT-RT finds significant register reuse opportunities, even increasing the lifetimes of registers in the PRF results in a lower total register file than the baseline due to the benefits of register sharing. Note that in our simulations we did not encounter a single situation where AT-RT used more registers than the baseline configuration despite using them for pipeline prefetching and keeping them alive for longer.

The complete view of the pipeline with all the new proposed structures is shown in Figure 7.

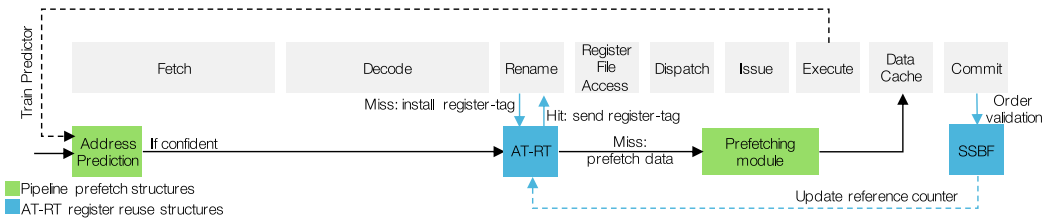

Fig. 7. Pipeline with the required structures for pipeline prefetching paired with our AT-RT register reuse structures.

Table 2. Gem5 Simulator Configuration

Frequency	3.6 GHz
IssueWidth/Ld,St Units	8/2,2
PRF (integer+float)	180+168 registers
SQ/LQ/IQ/ROB	56/72/50/224
iTLB/dTLB	512/512 fully-assoc
Address Predictor	Tagged 1K entry stride-predictor table with 2-bit confidence counter
Memory Order Valid	SSBF: Tagged 1K entry
DDT (only for IDist)	Tagged 1K entry table
AT-RT	512 entry table
Temp: 3520 Bytes	Tag: 39 bits,
Temp+Spat: 6208 Bytes	RegID: 8 bits,
	Data Size: 2 bits
	Ref-counter: 5 bits,
	Valid: 1 bit
Caches	L1I/L1D/L2/L3
Size	32 KB/32 KB/256 KB/8 MB
Latency	1c/2c/12c/38c
Associativity	8w/8w/8w/16w
DRAM	DDR 3, 1600 MHz, 64 bits

6 EVALUATION

6.1 Simulation and Modeling

We use 10 checkpoints (100 M instruction warming, 10 M instruction detailed simulation) for each SPEC2006 [9] application. We use gem5 [7] to simulate a large out-of-order X86_64 CPU (Intel Skylake-like 8-wide, 224 entry ROB, 72 entry LQ; full details in Table 2[1]). The first-level cache is dual-ported with pipelined loads and stores. For energy evaluations, we use CACTI [25] with a 22 nm technology node.[2]

We evaluate five configurations:

- **DLVP:** Address load predictor that prefetches the load data into the pipeline [39].[3]

[1]Note that cache latencies correspond only to the accesses latency. Load-to-use-latency is higher and includes issue and execution latency.

[2]For energy modeling, we model a 64 entry, 4-way set-associative dTLB to match the first-level TLB of the Intel Skylake architecture.

[3]For a fair comparison, our implementation of DLVP uses the same address predictor as the AT-RT configurations.

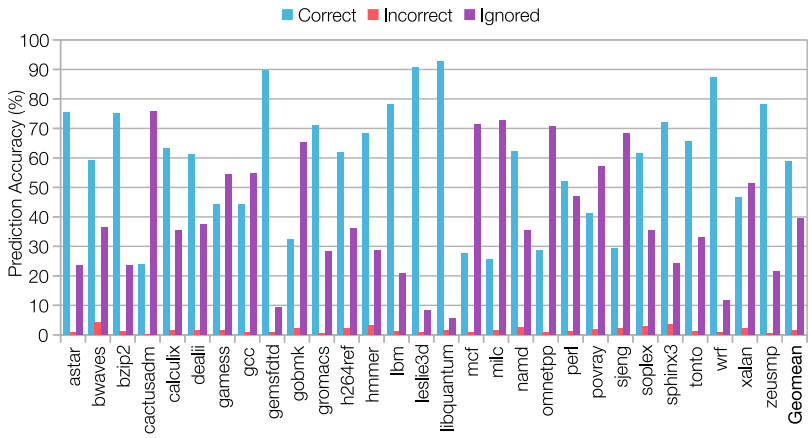

Fig. 8. Value prediction accuracy for all load addresses.

- **IDist:** Register reuse technique that uses instruction distance for load-to-load reuse [32].[4]
- **AT-RT (Temp):** DLVP pipeline prefetching + Our register reuse technique that uses predicted addresses to forward data (temporal reuse) and SSBF-based prefetch and register forwarding validation. This configuration is limited by the locality exposed by the LQ.
- **AT-RT (Temp+Spat):** DLVP pipeline prefetching + Our register reuse technique *with coalescing of loads* to the same 128-bit sub-array read-out (temporal and spatial reuse) and SSBF-based prefetch and register forwarding validation. This configuration is able to expose more locality than the one available in the LQ.
- **Baseline:** Standard pipeline with SSBF (to filter replays at commit), but no pipeline-prefetching and no register reuse.

6.2 Prediction Accuracy

The forwarding potential of AT-RT is a function of the available locality and the accuracy of the address predictor, with better predictors delivering better results. For this work, we choose a simple predictor, as described in Section 4.4. More advanced address predictors will simply result in more reuse opportunities and better performance/energy. Figure 8 shows the accuracy of our chosen stride address predictor. The stride predictor shows high coverage, with the majority of the benchmarks (17) able to accurately predict more than 60% of the addresses, and 8 benchmarks having an accuracy above 75%. Only 6 benchmarks have an accuracy below 40% with *cactusadm*, at 24% accuracy, being worst. On average, our simple predictor is able to accurately predict the address of 59% of the loads.

While low coverage limits the potential of AT-RT technique, incorrect address predictions can have a negative impact, since incorrectly forwarded loads must be replayed [1]. We found that most benchmarks (20) have misprediction ratios below 2% with an average of 1.6%.

6.3 Load-to-load Forwarding

Figure 9 compares how much load-to-load reuse the different configurations are able to take advantage of, and how much is exposed by the load queue (i.e., perfect reuse of all in-flight loads). IDist can forward 16% of the loads on average with the best benchmark being *lbm*, which has 41%

[4]While the standard ISRB does not filter any load instruction, for a fair comparison, we include our modified SSBF filter in the IDist configuration as well.

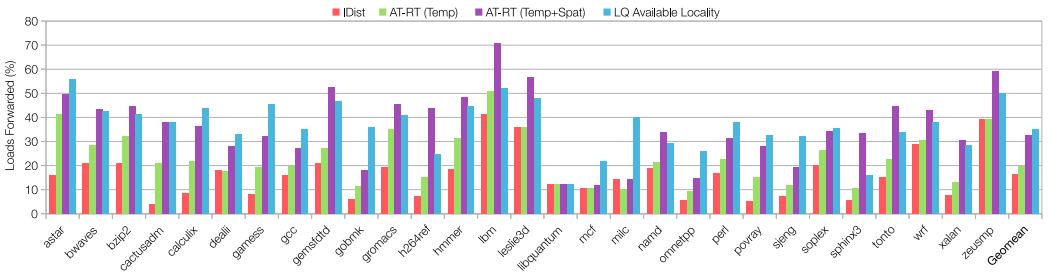

Fig. 9. Percentage of loads that have their data forwarded from the physical register file.

of its loads forwarded from the physical register file. Overall, IDist is able to extract 43% of the total load-to-load locality exposed in the load queue through the PRF.

AT-RT (Temp) is able to forward more data than IDist on 26 of 29 benchmarks. Of the three benchmarks where AT-RT performs worse than IDist, *milc* shows the highest difference, with the AT-RT forwarding 4 percentage points fewer loads than IDist. The remainder two benchmarks (*dealii* and *leslie3d*) fall behind by less than 1 percentage point (0.7, 0.08, respectively). However, AT-RT (Temp) shows the largest gains on *astar*, *cactusadm*, and *gromacs* with 25, 18, and 16 percentage point increase over IDist. Overall, AT-RT (Temp) is able to forward 20% of the loads (1.2× more than IDist) and is able to extract 56% (vs. 43% for IDist) of the load-to-load locality exposed in the load queue through the PRF.

The AT-RT (Temp+Spat) configuration fetches data in 128-bit blocks and installs it in multiple PRF entries when it is predicted to be used, and is thereby able to outperform both the IDist and AT-RT (Temp) across the entire benchmark suite. Benchmarks such as *h264ref*, *sphinx3*, and *xalan*, show an improvement of 1.9×, 2.2×, and 1.3× compared to AT-RT (temp), and 5×, 5.2×, and 3× compared to IDist.

Since AT-RT (Temp+Spat) takes advantage of the reduced register pressure from register forwarding to extend the lifetime of some registers, it is able to forward more data than what is exposed by the load queue window. On 14 of the benchmarks, AT-RT (Temp+Spat) is able surpass the total locality available in load-queue by extending register lifetimes. *Sphinx3* forwards 2.1× more loads than the best a load-queue-based forward predictor could, and *lbm* is able to forward 70% of its loads from the PRF. Overall AT-RT (Temp+Spat) reaches 93% of the load-to-load locality that the load queue exposes through the PRF. These results demonstrate that AT-RT is indeed able to achieve significantly improved spatial and temporal reuse over previous register-sharing techniques.

6.4 Memory Access Reduction

Since DLVP tries to prefetch every address-predicted load instruction, the higher the accuracy of the address predictor, the more prefetches and thus the more loads that need to be replayed for validation. Figure 10 shows the increase in L1 accesses by loads from DLVP compared to a standard non-pipeline-prefetching CPU. On average, DLVP executes 56% more loads accessing the L1 than the baseline and more than 90% more for *gemsfdtd*, *leslie3d,* and *libquantum*. The verification load could potentially be filtered by using SSBF (or ARB as proposed in APDP [15]), but this would only reduce the replays to match the baseline[5] (not shown in the graph), unlike IDist and AT-RT, which can reduce the number of L1 accesses below that of the baseline via register sharing.

[5]There would be a slight increase due to mispredicted prefetches, but those are rare, as it can be seen in Figure 8.

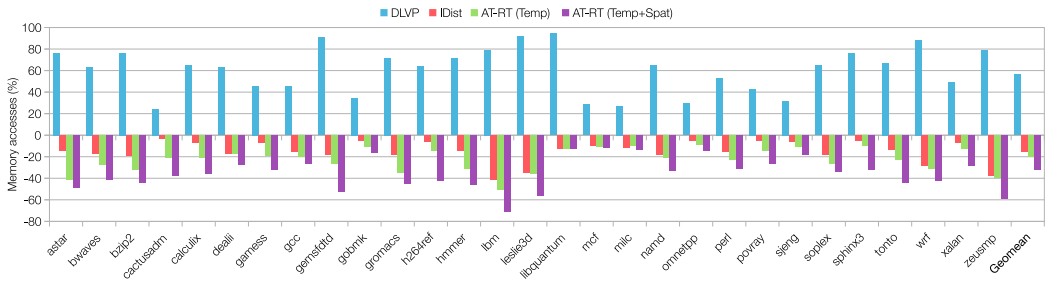

Fig. 10. Executed load instructions (L1 accesses by loads) compared to the baseline.

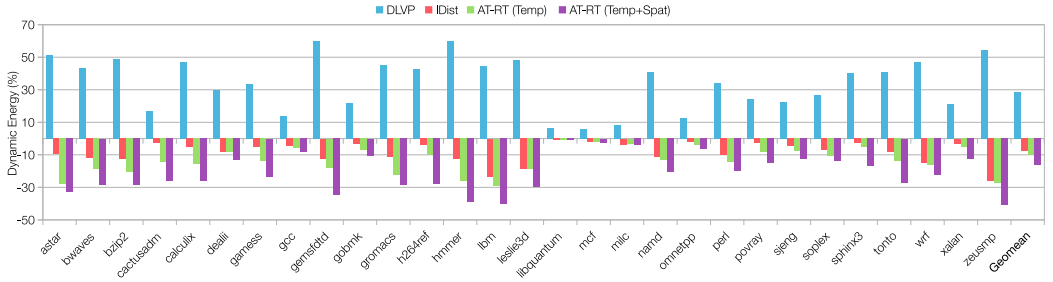

Fig. 11. L1 Dynamic energy of the different strategies compared to the baseline configuration. The AT-RT configurations include the dynamic energy overhead of accessing and updating the AT-RT table.

IDist (when extended with SSBF) and both AT-RT strategies can reduce the number of L1 load accesses to fewer than that of the baseline by not only avoiding DLVP's second verification load, but also avoiding sending the first prefetch to the L1 if the data is in the PRF. The number of L1 load accesses that can be filtered is directly related to the ability of the forwarding strategy to detect and take advantage of reuse. AT-RT (Temp) is able to reduce L1 load accesses by 20% on average, and up to 51% in *lbm*. This is an improvement over the 15% average L1 load access reduction of IDist with its highest reduction also being on *lbm* with 41%. Overall, AT-RT (Temp) shows a 1.3× improvement in avoided L1 accesses over IDist.

AT-RT (Temp+Spat) shows the largest improvement and eliminates 32% of the L1 load accesses compared to the baseline, a 2.1× improvement over IDist. *Lbm* is once again the benchmark with the largest reduction, by decreasing L1 load accesses by 71%.

The decrease in L1 accesses also reduces the L1 data cache dynamic energy, as shown in Figure 11. Note that there is not a one-to-one correspondence between dynamic energy reduction and memory accesses by filtered load instructions (Figure 10). This is because the dynamic energy is a function of both loads and stores, and the cache hit ratio, and the strategies evaluated in this article only affect load instructions. However, since most benchmarks are dominated by loads (70% or more on average) and have a high hit ratio, benchmarks with the highest percentage of filtered L1 accesses also show the highest reduction in dynamic energy. *Libquantum* and *gcc*, for example, stand out as exceptions due to the high number of cache misses and a low load-store instruction ratio (60%), respectively.

On average AT-RT (Temp) improves L1 dynamic energy by 10%, and up to 29% on the best benchmark (*lbm*). AT-RT (Temp+Spat) is able to improve on these results further by reducing L1 dynamic energy by 16% (1.6% total chip energy) compared to the baseline and up to 41% (4.1% total chip energy) on the best benchmark (*zeusmp*). Both strategies improve on IDist, which is only able

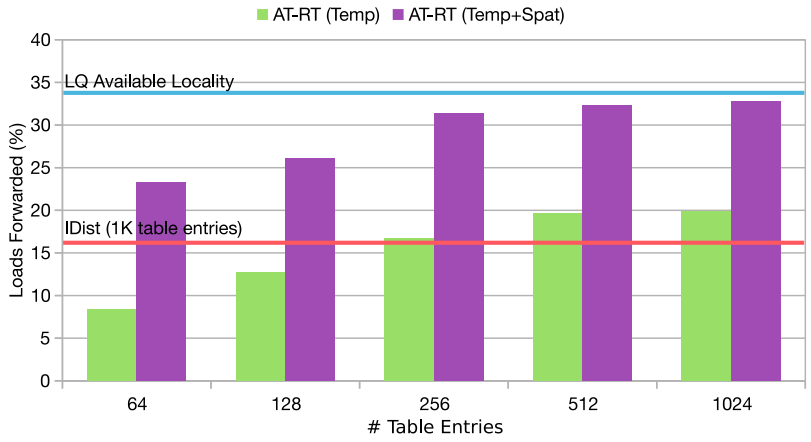

Fig. 12. Sensitivity to AT-RT table size. Percentage of loads that have their data forwarded from the physical register file, depending on the number of entries on AT-RT table. We evaluate a 512-entry AT-RT table in this work, but there is little loss in reuse at 256-entries.

to reduce L1 dynamic energy by 8% on average and 26% on the best benchmark (*zeusmp*), 0.8% and 2.6% total processor energy, respectively.

6.5 AT-RT Table Overhead

For our evaluation, we chose a large, multi-ported AT-RT table (512 entries, 6,208 B) to maximize reuse. This table is comparable to the other state-of-the-art register-sharing techniques and other baseline CPU components (e.g., memory order validation tables, value predictors). This table size represents about 0.5% of the core area (not including the L1 cache) and has an activation energy two orders of magnitude lower than an L1 access, as it reads very few bits and is direct-mapped. (These overheads are included in Figure 11.) Figure 12 explores the achievable reuse as a function of the AT-RT table size and shows that the table could be made half as large with little decrease in achievable reuse.

The largest source of complexity in this design comes from the need to simultaneously access the AT-RT table from multiple instructions during rename and commit. For our 8-wide rename and 6-wide commit pipeline, this could be as many as 2×8 (read and install if miss) + 6 (reference counter update) simultaneous accesses to the AT-RT table. However, AT-RT only needs to be accessed by load instructions, which account for only 18% of instructions fetched each cycle (32% worst case) in our simulations. Further, a lack of available AT-RT ports at rename simply reduces reuse opportunities, as loads can still be renamed as normal without checking for reuse. A lack of ports at commit would cause a commit stall. In our designs, we modeled an AT-RT table with 4 ports and observed no reuse opportunity losses or commit stalls.

6.6 Performance

Figure 13 compares the IPC improvement of the different strategies to the baseline configuration. IDist is able to improve performance by 3% on average and up to 7% compared to the baseline by taking advantage of the zero latency of forwarding data from the physical register file instead of the cache for reused registers. DLVP, as expected, provides a more substantial performance improvement, since it is able to prefetch a large percentage of loads (Figure 8). This means DLVP is able to provide more loads with a zero latency (through prefetching) than IDist (through reuse

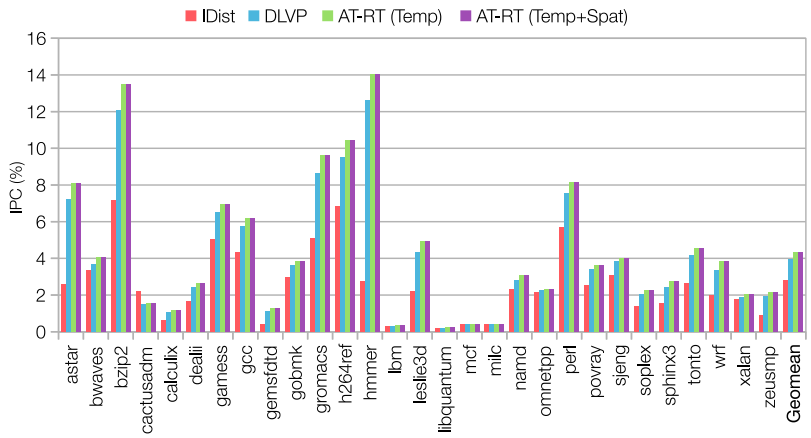

Fig. 13. Performance of the different strategies compared to the baseline configuration.

forwarding), and thus improve performance over the baseline by 4% on average and up to 13% on the best benchmark.

Since the AT-RT is able to forward more data through the PRF than IDist (shown in Figure 10), it should provide better performance benefit than IDist. As AT-RT uses the same address predictor as our implementation of DLVP, it also provides the same pipeline prefetching.

As expected, both AT-RT strategies achieve a similar performance benefit as DLVP, since they provide the same percentage of loads with zero load-to-use latency. Both AT-RT strategies do, however, significantly reduce the number of L1 accesses by loads compared to DLVP and thus also reduce L1 port pressure. This is because AT-RT can access prefetched loads directly from the PRF. The reduction in L1 pressure only translates into a marginal performance improvement: 0.4 percentage points improvement over DLVP and 1.4 percentage point on the best benchmark (*bzip2*). The small difference in IPC is due to the ability of the out-of-order core to deal with the extra L1 pressure of DLVP. In our simulations, the two L1 ports are able to handle most requests, and when they are not, the large issue width (8 instructions per cycle) and commit width of the core (6 instructions per cycle) is able to accommodate delayed validation loads (introduced by DLVP) with minimal impact. AT-RT (Temp+Spat) further reduces L1 accesses compared to the AT-RT (Temp) but the difference is too small for visible difference in IPC. These results demonstrate that AT-RT is able to achieve the full performance of pipeline prefetching with even better energy savings than register reuse.

6.7 Security

AT-RT employs speculation (prefetching and caching data in the PRF) and as such it is important to discuss how exposed our technique is to speculative side-channel attacks [19, 23], more specifically the ones that exploit changes made to the memory hierarchy under speculation. Our strategy involves two cache hierarchy levels, the L1 cache and the PRF that in our design is used as an extra cache level.

When it comes to the L1, the pipeline prefecther only speculatively fetches data that hits in the L1 cache. A cache miss will not trigger a fill request, and as such, it will never modify the contents of any cache in the hierarchy. Any activity by our prefetching/caching mechanism is completely invisible between execution contexts sharing the L1 or any other level in the memory hierarchy.

The contents of the AT-RT cache itself should also be invisible between contexts, since both the AT-RT table (that holds the tags) and the PRF (that holds the data) are private to the process/thread. On a context switch, the contents of the PRF is saved by the operating system before loading the PRF state of the new context (our solution makes no change to this behavior). We just add the extra step of clearing the AT-RT table during the switch. This is done for correctness, since it is functionally incorrect to keep the tags for data that no longer resides in the PRF.[6] As such, a new thread/process should always find a private AT-RT table and PRF, thus making it impossible to inspect the state of other contexts.

7 CONCLUSION

In this work, we have demonstrated that having address predictions early in the pipeline allows us to achieve the performance benefits of pipeline prefetching together with the energy benefits of register reuse, all without the overhead of requiring more data movement, data storage in the pipeline, or extra verification hardware/accesses.

To achieve this, we added address tags to the physical register file to allow us to search it for data returned by loads. This capability, combined with the early address prediction required for pipeline prefetching, allowed us to find data forwarding opportunities through the register file at rename with only the added cost of the register file tags and install coalesced data from reads to improve reuse. Further, we take advantage of the reduced register file pressure from register sharing to intentionally keep register file entries alive for longer than their consuming instructions to increase reuse. This allows us to exceed the spatial and temporal reuse available in previous register-sharing techniques. Finally, as we have addresses for the predicted prefetch addresses and can find data in the physical register file by address, we were able to re-use the existing memory order violation predictor hardware to validate forwarding and register reuse without the need for replaying the prefetch loads or additional hardware, as in previous work.

Using early address prediction throughout the pipeline enabled us to match the performance of state-of-the-art pipeline prefetching while reducing L1 dynamic energy by 16%, with only the cost of the register file tags (less than 0.5% total core area).

REFERENCES

[1] Ricardo Alves, Stefanos Kaxiras, and David Black-Schaffer. 2018. Dynamically disabling way-prediction to reduce instruction replay. In *Proceedings of the IEEE International Conference on Computer Design (ICCD'18)*.

[2] Ricardo Alves, Nikos Nikoleris, Stefanos Kaxiras, and David Black-Schaffer. 2017. Addressing energy challenges in filter caches. In *Proceedings of the IEEE International Symposium on High-performance Computer Architecture (SBAC-PAD'17)*. IEEE, 49–56.

[3] Ricardo Alves, Alberto Ros, David Black-Schaffer, and Stefanos Kaxiras. 2019. Filter caching for free: The untapped potential of the store-buffer. In *Proceedings of the 46th IEEE International Symposium on Computer Architecture*. ACM, 436–448.

[4] Steven Battle, Andrew D. Hilton, Mark Hempstead, and Amir Roth. 2012. Flexible register management using reference counting. In *Proceedings of the IEEE International Symposium on High-performance Computer Architecture*. IEEE, 1–12.

[5] Michael Bekerman, Stephan Jourdan, Ronny Ronen, Gilad Kirshenboim, Lihu Rappoport, Adi Yoaz, and Uri Weiser. 1999. Correlated load-address predictors. In *ACM SIGARCH Computer Architecture News*, Vol. 27. IEEE Computer Society, 54–63.

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

[6]Note that it is safe to clear the contents of AT-RT table at any point of the execution. A miss on the AT-RT table will only translate into a reduction in energy savings at worst. There are no correctness implications.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. DOI: https://doi.org/10.1145/2024716.2024718.

[8] George Z. Chrysos and Joel S. Emer. 1998. Memory dependence prediction using store sets. In *Proceedings of the 25th International Symposium on Computer Architecture*. IEEE, 142–153.

[9] Standard Performance Evaluation Corporation. 2006. SPEC CPU2006. Retrieved from: http://www.spec.org/cpu20066.

[10] Richard J. Eickemeyer and Stamatis Vassiliadis. 1993. A load-instruction unit for pipelined processors. *IBM J. Res. Devel.* 37, 4 (1993), 547–564.

[11] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta. 2005. Continuous optimization. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 86–97.

[12] Manoj Franklin and Gurindar S. Sohi. 1996. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Trans. Comput.* 45, 5 (1996), 552–571.

[13] Freddy Gabbay. 1996. *Speculative Execution Based on Value Prediction*. Technion-IIT, Department of Electrical Engineering.

[14] Roberto Giorgi and Paolo Bennati. 2007. Reducing leakage in power-saving capable caches for embedded systems by using a filter cache. In *Proceedings of the Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*. ACM, 97–104.

[15] José González and Antonio González. 1997. Speculative execution via address prediction and data prefetching. In *Proceedings of the International Conference on Supercomputing*. Citeseer, 196–203.

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

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

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

[19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher et al. 2019. Spectre attacks: Exploiting speculative execution. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'19)*. IEEE, 1–19.

[20] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. 2011. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *Proceedings of the International Conference on Computer-aided Design*. IEEE Press, 694–701.

[21] M. H. Lipasti. 1996. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*.

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

[23] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

[24] Andreas Moshovos, Scott E. Breach, Terani N. Vijaykumar, and Gurindar S. Sohi. 1997. Dynamic speculation and synchronization of data dependences. In *ACM SIGARCH Computer Architecture News*, Vol. 25. ACM, 181–193.

[25] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. *CACTI 6.0*. Technical Report HPL-2009-85. HP Labs.

[26] Soner Önder and Rajiv Gupta. 2001. Load and store reuse using register file contents. In *Proceedings of the 15th International Conference on Supercomputing*. ACM, 289–302.

[27] Lois Orosa, Rodolfo Azevedo, and Onur Mutlu. 2018. AVPP: Address-first value-next predictor with value prefetching for improving the efficiency of load value prediction. *ACM Trans. Archit. Code Optim.* 15, 4 (2018), 49.

[28] Arthur Perais, Fernando A. Endo, and André Seznec. 2016. Register sharing for equality prediction. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 4.

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

[30] Arthur Perais and André Seznec. 2014. Practical data value speculation for future high-end processors. In *Proceedings of the IEEE 20th International Symposium on High-performance Computer Architecture (HPCA'14)*. IEEE, 428–439.

[31] Arthur Perais and André Seznec. 2015. BeBoP: A cost effective predictor infrastructure for superscalar value prediction. In *Proceedings of the IEEE 21st International Symposium on High-performance Computer Architecture (HPCA'15)*. IEEE, 13–25.

[32] Arthur Perais and André Seznec. 2016. Cost effective physical register sharing. In *Proceedings of the IEEE International Symposium on High-performance Computer Architecture (HPCA'16)*. IEEE, 694–706.
[33] Arthur Perais, André Seznec, Pierre Michaud, Andreas Sembrant, and Erik Hagersten. 2015. Cost-effective speculative scheduling in high performance processors. In *Proceedings of the ACM/IEEE 42nd International Symposium on Computer Architecture (ISCA'15)*. IEEE, 247–259.
[34] Vlad Petric, Anne Bracy, and Amir Roth. 2002. Three extensions to register integration. In *Proceedings of the 35th IEEE/ACM International Symposium on Microarchitecture (MICRO'02)*. IEEE, 37–47.
[35] Vlad Petric, Tingting Sha, and Amir Roth. 2005. RENO: A rename-based instruction optimizer. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 98–109.
[36] Alberto Ros and Stefanos Kaxiras. 2018. The superfluous load queue. In *Proceedings of the 51st IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*. IEEE, 95–107.
[37] A. Roth. 2005. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 458–468.
[38] Amir Roth. 2008. Physical register reference counting. *IEEE Comput. Archit. Lett.* 7, 1 (2008), 9–12.
[39] Rami Sheikh, Harold W. Cain, and Raguram Damodaran. 2017. Load value prediction via path-based address prediction: Avoiding mispredictions due to conflicting stores. In *Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture*. ACM, 423–435.
[40] Avinash Sodani and Gurindar S. Sohi. 1997. Dynamic instruction reuse. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA'97)*.
[41] Nathan Tuck and Dean M. Tullsen. 2005. Multithreaded value prediction. In *Proceedings of the 11th International Symposium on High-performance Computer Architecture*. IEEE, 5–15.
[42] Kai Wang and Manoj Franklin. 1997. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, 281–290.

Received December 2020; revised March 2021; accepted March 2021