2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)

Boosting Store Buffer Efficiency with Store-Prefetch Bursts

Juan M. Cebrian
Computer Engineering Department
University of Murcia
Murcia, Spain
Email: jcebrian@um.es

Stefanos Kaxiras
Department of Information Technology
Uppsala University
Uppsala, Sweden
Email: stefanos.kaxiras@it.uu.se

Alberto Ros
Computer Engineering Department
University of Murcia
Murcia, Spain
Email: aros@ditec.um.es

***Abstract*—Virtually all processors today employ a store buffer (SB) to hide store latency. However, when the store buffer is full, store latency is exposed to the processor causing pipeline stalls. The default strategies to mitigate these stalls are to issue *prefetch* for ownership requests when store instructions commit and to continuously increase the store buffer size. While these strategies considerably increase memory-level parallelism for stores, there are still applications that suffer deeply from stalls caused by the store buffer. Even worse, store-buffer induced stalls increase considerably when simultaneous multi-threading is enabled, as the store buffer is statically partitioned among the threads.**

In this paper, we propose a highly selective and very aggressive prefetching strategy to minimize store-buffer induced stalls. Our proposal, Store-Prefetch Burst (SPB), is based on the following insights: i) the majority of store-buffer induced stalls are caused by a few stores; ii) the access pattern of such stores are easily predictable; and iii) the latency of the stores is not commonly hidden by standard cache prefetchers, as hiding their latency would require tremendous prefetch aggressiveness. SPB accurately detects contiguous store-access patterns (requiring just 67 bits of storage) and prefetches the remaining memory blocks of the accessed page in a single burst request to the L1 controller. SPB matches the performance of a 1024-entry SB implementation on a 56-entry SB (i.e., Skylake architecture). For a 14-entry SB (e.g., running four logical cores), it achieves 95.0% of that *ideal* performance, on average, for SPEC CPU 2017. Additionally, a 20-entry store buffer that incorporates SPB achieves the average performance of a standard 56-entry store buffer.

I. Introduction

Memory latency continues to limit the performance of modern out-of-order cores, despite the efforts to hide load and store latency. Load latency is hidden thanks to two well-known techniques [13]: out-of-order execution of loads and prefetching mechanisms. Out of order execution of loads is achieved either by relaxing the consistency model provided by the system (e.g., ARM and IBM Power), or through speculative execution mechanisms when strong consistency guarantees are provided (e.g., Intel's and AMD's total store order – TSO) [13]. Regarding prefetching, considerable research efforts have been dedicated to develop effective strategies for the different cache levels across the memory hierarchy [2]–[4], [16], [17], [19], [21], [28]. Store latency has been traditionally hidden by the store buffer (SB), which allow stores to commit as long as the SB is not full. Stores are delayed in the SB while other instructions (e.g., loads) can commit, effectively relaxing the consistency model of the system (e.g., x86-TSO [25]). However, when the SB fills (e.g., due to a cache miss), the processor stalls. This scenario is common on store bursts, shifting the processor bottleneck to the SB [9], [12], [23].

Research efforts regarding the efficient management of stores in TSO focus on both store coalescing and performing stores out of order without affecting consistency, but that comes at the cost of increased complexity [24], [31]. Nevertheless, little effort has been directed at quantifying the impact of store prefetching, usually by blindly applying the same load prefetching strategies for stores. Specific techniques for stores are almost absent in the literature, and focus mostly on requesting the ownership (i.e., write permission) of the memory block to be written before the write takes place but after the actual store address is computed [13], [29].

Store prefetching [13], [29] is a fundamental technique for enabling memory-level parallelism (MLP) for stores (more specifically, store-level parallelism – SLP), especially in processors that enforce the store→store order as the write-permission request can safely resolve out of order. Intel's approach, as far as we can tell from published material, consists of issuing the request for ownership when the store instruction commits [15]. SLP increases using this technique, but only for the stores that fit in the SB, which correspond to only a few different memory blocks.

Even with the currently implemented store prefetching strategies, the SB becomes a critical component in memory-bound applications, reaching up to a third of the CPU stall cycles on database applications [22]. Indeed, the SB size keeps growing to prevent stalls, but its maximum size is limited by CAM (content-addressable memory) access latency, since every load must associatively search the SB for a matching store [11]. For example, the number of entries of the SB in Intel processors has increased from 32 to 56 in just 10 years, which is significant considering the cost involved in enlarging CAM structures. In fact, SB stalls are so critical that Intel has

This work has been supported by the Spanish Ministerio de Economía, Industria y Competitividad – Agencia Estatal de Investigación (grant ERC2018-092826) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134), the Vetenskapsradet project 2018-05254 and by the European joint Effort toward a Highly Productive Programming Environment for Heterogeneous Exascale Computing (EPEEC) (grant No 801051).

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00054

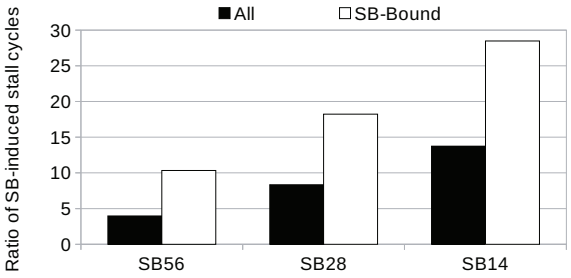


Fig. 1: Ratio of stall cycles due to a full SB for SPEC CPU 2017. "*All*" represents the average of all applications in the benchmark suite while "*SB-Bound*" represents those applications that exhibit more than 2% SB-induced stalls for a 56-entry SB. Simulation details are given in Section V.

specifically classified them in their Top-Down model under the memory-bound classification [33].

Furthermore, on processors that support simultaneous multi-threading (SMT), the effective size of the SB is divided by the number of hardware threads as the SB is statically partitioned across threads (Section 2.6.9 of Intel's optimization manual [15]). This partitioning is related to the consistency model, and in particular, to *store atomicity semantics* as dictated by the read-own-write-early-multiple-copy atomic model (rMCA) that is followed in actual x86-TSO implementations [1], [30]. SMT is usually enabled by default, so when not used some of the processor resources are underutilized. Figure 1 shows how the percentage of SB-induced stalls increases as the size of the SB is reduced from 56 entries to one fourth (14 entries), as it would happen in a SMT-4 processor.

In this work we propose Store Prefetch Burst (SPB), a prefetching technique tailored to store instructions, that effectively removes SB-induced stalls. Our proposal is based on three insights: i) only a few store instructions are responsible for the majority of SB-induced stalls; ii) the access pattern of the store instructions blocking the SB is easy to predict as they frequently correspond to sequential memory-block accesses (often as a consequence of data movement code); and iii) the stores require a very aggressive prefetch degree, i.e, a large amount of consecutive memory blocks have to be prefetched, in order to hide their write latency. The latter is the reason why standard prefetchers are unable to transform the majority of store misses into hits, but, at best, into a limited number of *late prefetches*, i.e., prefetches that should have been issued earlier to hide the full latency.

SPB detects contiguous store access patterns in the SB and predicts that the pattern will continue for instructions that are not currently in the SB (due to size limits). At that point, it triggers a *store-prefetch burst* in the L1 controller that requests write permission for all the remaining memory blocks within the current page. This prediction, differently from non-predictive techniques [13], [29], allows SLP to be exploited outside the code scope delimited by the SB.

Although in this work we present SPB on top of a TSO-like SB, it can be generally applicable to SBs that relax the store→store order, e.g., the ones implemented in ARM and IBM Power architectures.

Our results show that a 56-entry SB, with the default prefetch policy, is maximally-sized for today's processors to yield a 98.1% of the performance of an ideal (no-stalls) SB implementation. With a simple implementation and minimal hardware requirements (67 bits of storage), SPB achieves a virtually identical performance than the ideal SB implementation (100%). More importantly, SPB excels when SB resources are limited, which makes it essential for both SMT and energy-efficient designs. In fact, when halving the SB size (i.e., the per-thread size of the SB if SMT-2 was enabled), the default store prefetching strategy only achieves 93.6% of the ideal SB performance, while SPB reaches 98.9%. For a 14-entry SB (i.e., the per-thread size of the SB with 4 SMT threads, as in Intel's Knights Landing, IBM Power 9 or the rumored AMD Zen 3 family), the default prefetching strategy achieves 85.9% of the potential performance, while SPB achieves 95.4%, on average, for all SPEC CPU 2017 applications (including those that are not limited by SB-induced stalls). Alternatively, SPB offers an opportunity to reduce the SB size for energy-efficient implementations. Indeed, a 20-entry SB with SPB achieves the same average performance than a standard 56-entry SB.

The main contributions of this paper are:

- First comparison of store prefetchers [13], [29], after an accurate implementation in gem5.
- Deep analysis of the reasons for SB-induced stalls.
- A simple and efficient proposal to alleviate SB stalls.
- Analysis with standard and small SB sizes, showing that SPB approaches ideal SB performance in all cases.
- Analysis of SPB with aggressive prefetching schemes, showing that SPB is orthogonal to other prefetching strategies, proving to be a good addition for improved accuracy with minimal hardware resource requirements.

II. Background

SB stalls can be critical when running memory-bound applications, especially those that require constant data movements (e.g., databases, video compression, rendering). There are three main alternatives to reduce SB-induced stalls: to increase the effective size of the SB, to minimize the latency of the stores via prefetching, and to allow stores to execute out-of-order. The pros and cons of the first alternative have been elaborated in the introduction and the last alternative will be discussed in the related work. This section focuses therefore on existing dedicated prefetching techniques for stores, which we will use to compare our proposal with.

Two main alternatives for store prefetching initiated by the processor can be found in the literature. In the first one, proposed by Gharachorloo *et al.* [13], the prefetch for ownership request is initiated as soon as the address of a store instruction is computed. We will refer to this approach as *at-execute* as addresses are commonly computed at the execute pipeline stage. In the second one [15], [29], the prefetch is initiated once the store instruction commits and it is inserted in the SB. We name this alternative *at-commit*. In both cases

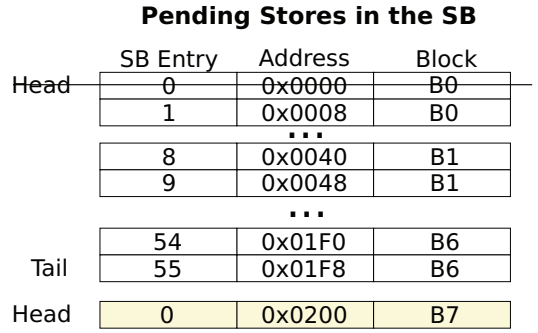


Fig. 2: Accesses to consecutive blocks in a 56-entry SB

we assume that the prefetch will allocate the block with write permission in the L1 cache. Both techniques rely on the actual address to perform the prefetch.

At-execute is the earliest time that a non-predicted prefetch can be issued. The main advantage of this approach is that it increases the chances that the ownership is ready at the L1 cache when the store reaches the head of the SB. However, this policy issues prefetches when the store is still speculative since stores may be later squashed for a several reasons, e.g., stores belong to an incorrect path of execution due to branch miss-prediction. This means that both energy and cache resources may be wasted by moving unnecessary data through the memory hierarchy. In addition, bringing the ownership of the data to the L1 cache too early can lead to other accesses evicting the block before the store performs.

On the other hand, the store prefetch requests sent at-commit are not speculative, and it is certain that the store will be performed (when it reaches the head of the SB). Therefore, only useful data is stored in cache with this policy. The drawback is that prefetching at commit can increase the number of late prefetches when comparing to at-execute, thus causing more stalls in the processor.

Both techniques, at-execute and at-commit, favor MLP, allowing to bring more memory blocks in parallel to L1 cache. Unfortunately, they limit the parallelism. At-execute can just exploit parallelism for stores that have been executed and have not yet been performed. At-commit exploits parallelism for a lower number of stores, the ones committed but not performed yet. In general, the window of opportunity to hide miss latency is small for both approaches since prefetches are issued close to the end of the instruction's life cycle. In contrast, we expand the prefetch window by accurately predicting future store addresses.

III. Motivation

A. Limitations of prefetching techniques

When no prefetching mechanism is employed stores are serialized making the SB a serious bottleneck. Figure 2 shows a common access pattern in which a large number of stores perform 8-byte writes to contiguous addresses. If the store at the head of the SB misses (block B0), the SB may fill up, thus, stalling the processor pipeline.

Prefetching techniques for both loads and stores are usually applied to all cache levels. Cache prefetchers can easily predict a stride access pattern, bringing one or more memory blocks depending on an adaptive throttling mechanism [28]. However, the prefetching degree of the lower cache levels is usually low, since, in the event of a prefetch misprediction, the cache can be seriously polluted [14]. For example, an L1 prefetcher may only prefetch the next block when the cache is accessed [15]. In that case, and following the example in Figure 2, when the store in entry 0 performs (block B0), it will just prefetch block B1. Block B2 will only be prefetched when the store in entry 8 (block B1) performs, which will be after bringing the block B1 from memory to L1.

Store prefetch techniques such as at-execute and at-commit further improve MLP, since several blocks are prefetched to the L1 cache in parallel. Indeed, all the memory blocks required by stores in the SB can be prefetched in parallel, as long as hardware resources allow it. For example, assuming the same example of a store instruction in a tight loop that generates sequential stores to memory locations of 8 bytes, an at-commit policy for a 56-entry SB will prefetch 7 blocks, assuming 64-byte blocks (Figure 2). When the oldest entry in the SB is freed, a store instruction that uses block B7 is inserted. At that point an at-commit prefetcher will request block B7 and, when the prefetch request succeeds, the L1 prefetcher will also bring block B8, but that is as far as it will go. The prefetching degree of at-commit and at-execute can be higher than the one employed by prediction-based cache prefetchers. However, assuming a best case scenario where all stores hit in the L1 cache, only one entry from the SB will be released every cycle. This means that the prefetcher will bring a new block to L1 every 8 cycles (we assume 8 writes per memory block in the example), going back to serialization and missing MLP. In essence, having a big SB is especially useful on sparse codes, but of limited gain on dense codes with store bursts.

B. Characterization of SB-induced stalls

We performed a detailed analysis of SB-bound applications which revealed that: i) a limited number of program counters (PCs) caused most SB-induced stall cycles, ii) the code sections where these PCs belong are mostly in the operating system and system libraries, and iii) these codes write large chunks of contiguous data.

In effect, Figure 3 shows that some of the SB-bound applications spend most time stalled on stores that belong to library calls (memcpy, memset, calloc) or the operating system (clear_page_orig[1]). All these functions produce large store bursts when applied to big data structures, causing stalls in the pipeline once the SB fills up.

On the other hand, for applications like *deepsjeng* or *roms*, most SB-induced stall cycles are produced by PCs from the application itself. Indeed, store bursts are created by manually moving data between data structures, or resetting a memory allocation to constant values (e.g., 0).

[1]This function is called by the OS every time a page is assigned to user code. It sets a memory page to zero for security reasons.

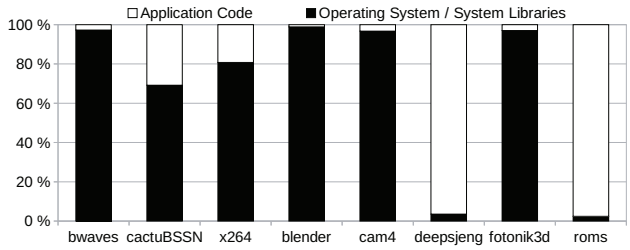


Fig. 3: Location of stores causing SB-induced stalls for SB-bound applications in SPEC CPU 2017

In addition, it is important to note that applications that rely on STD containers (vectors, maps, etc), would be constantly moving data around to resize the containers when required. This issue will be exacerbated in big data applications, where most of the data footprint does not fit in the last level cache. The same will apply to high-level development environments that manage memory transparently at runtime, such as Java or C#, if they need to reallocate data or use their garbage collection while the user application is running.

IV. Store-Prefetch Bursts

In this section we describe the concept and implementation of SPB, based on the motivation and observations presented in the previous section. The goal of SPB is to detect store bursts to contiguous addresses and prefetch as many blocks as possible within the current memory page.[2]

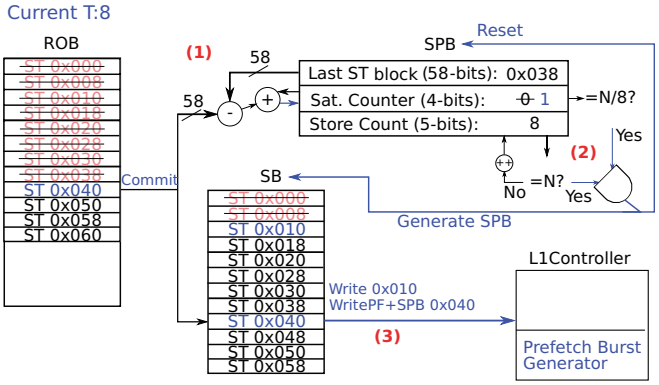
A first approach could be to detect a contiguous store access pattern based on the memory addresses of stores. However, we observed that detecting such contiguous access pattern is very restrictive and misses many opportunities. The reason is that store addresses are indeed shuffled by the compiler (e.g., reordering after a loop unrolling), and even though memory blocks are fully accessed, addresses themselves do not follow a contiguous accesses pattern.

In order to capture complex patterns, such as shuffling and interleaving, while keeping an elegant implementation, we propose to detect accesses to contiguous memory blocks within a certain time frame (or number of stores). This allows SBP to tolerate a certain degree of interleaving and shuffling, as long as all accesses still map to the same memory block. Once a contiguous access pattern is detected, SPB predicts that this pattern will repeat for the whole memory page. Then SPB issues a prefetch request for write permissions to the L1 controller for the remaining blocks in the page (forwards).

A. Micro-architecture

The implementation of SPB requires minimal hardware modifications and resources. Figure 4-(top) shows a diagram of the proposed micro-architectural extensions. SPB employs just three registers to detect when to trigger the prefetch burst (a total of 67 bits). The first register, *last block*, stores the address of the block being written by the last store that committed (58 bits). The second register is a *saturated counter* that tracks consecutive blocks (4 bits). The last register, *store count*, keeps the number of stores considered by SPB (5 bits).

[2]We did not explore in this work prefetching beyond page boundaries despite our prefetcher can work with virtual addresses and overcome the limitation of cache-level prefetchers: consecutive virtual pages could not map to consecutive physical pages.



	ROB	SPB	SB	L1Controller
T0:	Commit ST 0x000	Last = 0x00 Sat. = 0 St Count = 1	TSOBlock = 1 Write 0x000	Miss (0x000) I->IM: Getx
T1:	Commit ST 0x008	Last = 0x00 Sat. = 0 St Count = 2	TSOBlock == 1 WritePF 0x008	IM->IM: PopReq.
T2:	Commit ST 0x010	Last = 0x00 Sat. = 0 St Count = 3	TSOBlock == 1 WritePF 0x010	IM->IM: PopReq.
T3:	Commit ST 0x018	Last = 0x00 Sat. = 0 St Count = 4	TSOBlock == 1 WritePF 0x018	IM->IM: PopReq.
T4:	Commit ST 0x020	Last = 0x00 Sat. = 0 St Count = 5	TSOBlock == 1 WritePF 0x020	IM->IM: PopReq.
T5:	Commit ST 0x028	Last = 0x00 Sat. = 0 St Count = 6	TSOBlock = 0 WritePF 0x028	IM->M: PopReq.
T6:	Commit ST 0x030	Last = 0x00 Sat. = 0 St Count = 7	TSOBlock = 1 Write 0x008 WritePF 0x030	Hit (0x008) M->M M->M: PopReq.
T7:	Commit ST 0x038	Last = 0x00 Sat. = 0 St Count = 8	TSOBlock = 0 WritePF 0x038	M->M: PopReq.
T8:	Commit ST 0x040	Last = 0x01 Sat. = 1 -> 0 St Count = 0	TSOBlock = 1 Write 0x010 WritePF+SPB 0x040	Hit (0x010) M->M Misses (0x040+) I->PF_IM: GetPFx I->PF_IM: GetPFx ..... I->PF_IM: GetPFx

Fig. 4: SPB micro-architecture (top) and running example (bottom). I, M, IM, PF_IM, Getx, GetPFx and PopReq. are gem5 MESI coherence protocol states and messages.

When a store commits, SPB gets the address of the block to be written by the store (omitting the six least significant bits of the target memory address for 64-byte blocks) and computes the difference with respect to the block address of the last committed store (Figure 4-(1)). If the difference is 0, both stores access the same block and the saturated counter is not modified. If the difference is 1, the stores access consecutive blocks, and the saturated counter is increased. In any other case, the saturated counter is reset. After each store computes the difference, the last block register is updated with the current block address and the store count is increased.

Every *N* stores (where *N* is a configurable parameter), we check the saturation counter (Figure 4-(2)). If the saturation counter shows that we have accessed *N/8* blocks (as a 64-byte block can have 8 consecutive 8-byte stores), we infer that we are storing data to consecutive locations and we predict that there will be a store burst. It is relatively simple for SPB to prefetch backward store bursts (e.g., to prefetch data from the stack). However, we found no evidence that backward store bursts cause SB stalls, so this extension is not considered in this work, as it would not provide perceptible performance improvements for the evaluated applications.

B. Running example

Figure 4-(bottom) shows a running example considering the consecutive access pattern of 64-bit stores used in the motivation section. To simplify the example, we assume that the processor commits one instruction per cycle, that all stores miss in L1, and that the L1 and L2 latencies are 1 and 4 cycles, respectively. If SPB is configured to check the saturation counter every 8 stores (N = 8), the differences computed by SPB would be 0, 0, 0, 0, 0, 0, 0, 0, and 1 (Figure 4-(1)). This means that we have accessed two different blocks in the last 8 stores, and therefore, the saturation counter is incremented by 1. During each cycle the SB continues to send write prefetch requests using the default at-commit policy, that will be discarded (PopReq) when they find that the requested block is already in L1. Per-cycle coherence protocol state transitions and messages are also shown in Figure 4. After 8 stores (T8), the saturation counter is compared to (8/8, Figure 4-(2)). Since the results match, SPB sends a prefetch burst request to the L1 controller (Figure 4-(3)).

C. Sensitivity to parameters

We performed a sensitivity analysis with respect to the choice of *N*. We found that the optimal *N* value varies depending on the SB size, more specifically, *optimal N* is 48 for a SB of 14 entries, 24 for an SB of 28 entries, and 48 for a SB of 56 entries. In general, values between 24 and 48 lead to high performance. For our evaluation, we chose a value of 48 for *N* as the variability of the results for the 28-entry SB is minimal, for *N* between 24 and 48.

We also considered an SPB variant that dynamically adapts to different data sizes, not systematically assuming 8-byte stores. That is, the saturation counter is not tested against *N/8*, but to a dynamically adjusted threshold *N/S*, where *S* depends on the data size of the stores in that window. Our results showed that this variant performs worse than the simple SPB presented in the previous subsection, due to adaptation hysteresis and lost opportunity.

D. Discussion: Software alternative

An alternative to SPB would be to perform aggressive software prefetching in code sections where the programmer thinks that there would be store bursts (e.g., system libraries or kernel functions).

However, there are several reasons why a hardware implementation is a better choice. First, programmers may not rely on library calls, performing the memory copies manually with "for" loops (e.g., *deepsjeng* and *roms*). Second, introducing many software prefetching instructions would fill the pipeline with "useless" instructions (e.g., 64 prefetch instructions per memory page). Third, software prefetches will not have any effect if they entail page faults. Fourth, software prefetchers would bring the block to cache, but not necessarily this write permission.

TABLE I: Configuration parameters

Chip details	
Core	1 and 8 out-of-order cores, 2.0GHz
Core details	
Fetch, decode, rename width	4 instructions
Dispatch, issue, commit width	4 instructions
TLB	8 way, 1KB
Branch predictor	L-TAGE 64KB
Branch target buffer	8K+8K entries
Fetch buffer, decode buffer	16B, 56-μops
Fetch, load and store queues	32 entries, 72 entries, 56 entries
Physical registers	180 integer + 180 floating point
Issue queue, re-order buffer	97 entries, 224 entries
Functional units	1 Int ALU + 3 Int/FP/SIMD ALU
Instruction latencies (int)	add (1c.), mul (4c.), div (22c.)
Instruction latencies (fp)	add (5c.), mul (5c.), div (22c.)
L1 instruction cache	32KB, 8-way, latency: 1 cycle
L1 data cache	32KB, 8-way, latency: 4 cycle
L1 prefetcher	Stream prefetcher (stride)
L2 cache	1MB, 16-way, latency: 14 cycle
L3 unified cache	16MB, 16-way, latency:36 cycle
MSHR entries	64 per cache (per bank at L3)

V. Methodology

We employ the widely used *gem5* [6] simulator modeling a x86 full-system environment. We simulate single- and multi-core processors using the detailed out-of-order CPU and memory hierarchy of gem5. Table I summarizes the main simulation parameters The simulated system runs Ubuntu 16.04 with Linux kernel 4.9.4. We developed extensions to simulate an x86 Skylake-X processor. Execution and issue latencies are modeled as measured on real hardware by Fog [10]. We modified gem5 to support pipelined L1 accesses for stores.

Energy consumption is evaluated with McPAT [18] using a process technology of 22nm (minimum available in the current version), a voltage of 0.6V and the default clock gating scheme. We incorporate the changes suggested by Xi et al. [32] to improve the accuracy of the models. We model the extra accesses to the L1 and the prefetch requests generated by SPB. The cost of the registers introduced by SPB and the logic to calculate the block difference is negligible compared to the rest of the structures in the core.

gem5 does not include a specific store prefetching (but a generic L1 stream prefetcher for both loads and stores). The lack of a dedicated store prefetcher causes serialization of the stores, and leads to sub-optimal performance. We have implemented an at-commit prefetch strategy [15], [29], according to the specification from the Intel optimization manual (Sec. 2.1.5.1), showing speedups of 15%, on average, for SPEC CPU 2017 and the system configuration described in Table I. We employ the at-commit implementation as our baseline

system as it is the default option available in real hardware. We also implemented the at-execute prefetch policy [13] as an academic alternative. Finally, we also compare our results against an ideal SB implementation (Ideal) that has no stalls due to SB capacity issues and all blocks in the SB are prefetched for permissions in parallel.

In addition to the three different prefetching strategies (at-commit, at-execute and SPB), we consider three different SB sizes in our evaluation: a 56-entry SB (SB56), a 28-entry SB (SB28), and a 14-entry SB (SB14).

We run the whole SPEC CPU 2017, compiled using GCC 5.5 with -O2 optimization flags. Statistics are gathered for the Region of Interest (ROI) of the benchmarks. The ROI begins *after* the initialization phase of the application and ends before any final output. We simulate 2 billion instructions inside the ROI, after a brief warm-up of the caches for 100 million cycles within the ROI.

We define as SB-bound applications those that show more than 2% of SB-induced stalls for our baseline configuration: *bwaves*, *cactuBSSN*, *x264*, *blender*, *cam4*, *deepsjeng*, *fotonik3d*, and *roms*. For the sake of clarity we only show per-application results for SB-bound benchmarks. Nevertheless, all figures include a bar that represents the geometric mean for all (ALL) benchmarks in the SPEC CPU 2017 suite, as well as a bar that isolates only the SB-bound (SB-BOUND) benchmarks.

In addition, we run all the applications from the PARSEC [5] multi-threaded benchmark suite with eight threads and *simlarge* inputs (except *freqmine* and *raytrace* that did not run correctly under gem5). We measure performance within the region of interest (ROI), that is, all instructions executed after initialization and before output. Statistics are gathered after 100 million cycles within the ROI to warm up the caches. The SB-bound applications in PARSEC are *bodytrack*, *dedup*, *ferret*, and *x264*, according to our criteria of more than 2% SB-induced stalls in the baseline configuration.

VI. Evaluation

In this section we show in detail how SPB outperforms all previous prefetching strategies in terms of both execution time and energy consumption. In addition, we show how all prefetching approaches behave regarding the reduction of SB-induced stall cycles at issue, as well as how reducing SB stalls affects overall issue stalls due to other resources (e.g., registers, reorder buffer, load queue, etc). Next, we show how SPB behaves as a prefetcher, showing the success rates as well as the increment in prefetch network traffic and L1 accesses (to tags). Finally, we show how this increment in traffic and L1 accesses affects the performance of the applications. To do so, we rely on Intel's Top-Down model information [33]. More specifically we will focus on execution stalls while there are L1D misses pending. This statistic is used to define the level of memory-boundness of the application [20].

A. Performance and energy

Figure 5 shows the execution time normalized to an *ideal* 1024-entry SB for all the evaluated prefetching strategies and

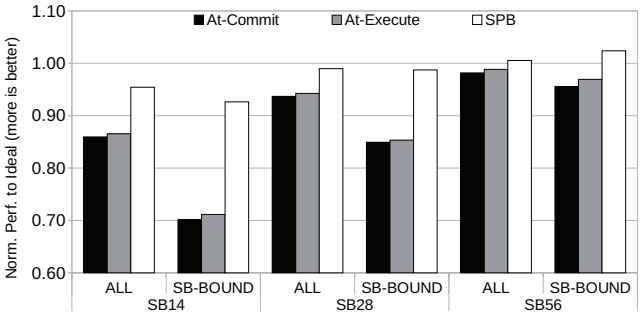


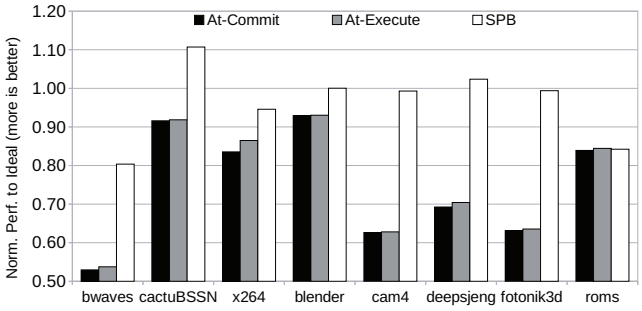
Fig. 5: Norm. performance to Ideal for different SB sizes

different SB sizes. For a SB of 56 entries, our proposal achieves 100.5% of the performance of an ideal SB. This provides an additional 2% performance improvement over Intel's strategy (at-commit - 98.1%). The performance gap between at-commit and SPB increases to 6.8% for SB-bound applications, where SPB also achieves 102.3% of the the ideal SB performance. This super-linear speedup comes from the predictive nature of SPB. Although initially unintended, SPB brings cache blocks to L1D that are also used by loads in the application. This reduces the average wait time of loads (as we will see later in Figure 14), and that results in a reduction of misspeculated instructions. Indeed, up to 10% misspeculated instructions are no longer executed, since branches are resolved faster with SPB.

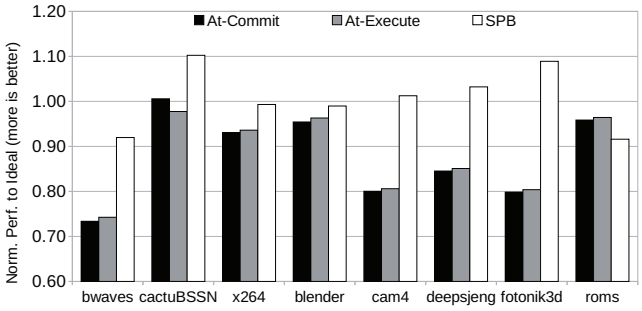
When we move to SB28, SPB can provide additional 5.2% performance when compared to at-commit for all SPEC CPU 2017. For this SB size we manage to achieve 98.9% of the ideal SB performance. If we focus only on SB-bound applications, SPB outperforms at-commit by 13.80% (98.7% of ideal). Finally, in the case of SB14, the use of advanced prefetching strategies becomes critical. In this scenario, the at-commit strategy barely reaches 85.9% or the ideal SB performance, 70.1% if only SB-bound applications are considered. On the other hand, SPB raises those numbers to 95.4% and 92.6%, respectively. This translates into an improvement of around 9.5% for all the SPEC CPU 2017, and 22,5% for SB-bound applications.

The advantages of SPB are, therefore, clear. SPB allows for smaller SB sizes while keeping performance levels much closer to an ideal implementation. Indeed, a 20-entry SB with SPB suffices to achieve the same average performance as a standard 56-entry SB.

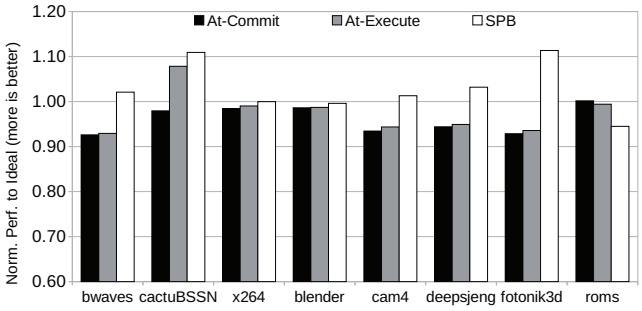
Figure 6 shows the per-benchmark performance improvements for SB-bound applications. For cactuBSSN, blender, cam4, deepsjeng and fotonik3d, the SB can be shrunken down to 14 entries without major performance penalties. However, for the rest of applications, having a 14-entry SB will pose a serious performance penalty. There are several applications that show better performance for SPB than the ideal SB. As mentioned previously, this is due to a beneficial side effect on L1 load hit rate for some configurations. In roms the compiler has interleaved many stores from loop unrolling,



(a) 14-entry SB



(b) 28-entry SB



(c) 56-entry SB

Fig. 6: Performance of SB-Bound applications normalized to an ideal SB

and prefetches by SPB produce additional L1 misses (conflict misses more specifically).

Finally, Figure 7 shows the effects of SPB in cache dynamic energy consumption, total core dynamic energy consumption and overall core energy consumption (dynamic + static). SPB slightly reduces the dynamic energy consumed by all cache levels. While SPB increases prefetching traffic and block replacements (between 0.4% and 1.1%), there is also a significant reduction on L1D cache accesses coming from misspeculated instructions. Given that we are executing less misspeculated instructions, the overall dynamic energy is reduced. Therefore, the net energy savings for all SB sizes are positive. Moreover, the reduction in total leakage energy due to the performance improvements given by SPB offsets the increase in cache dynamic energy, thus further reducing energy consumption. More specifically, net savings for SB sizes of 14,

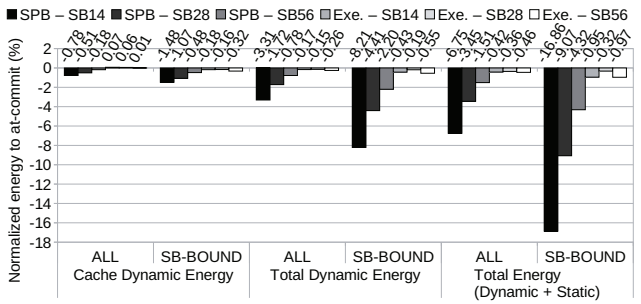


Fig. 7: Normalized energy to at-commit (less is better). Breakdown in cache dynamic energy (L1+L2+L3), total core dynamic energy and total energy (dynamic + static)

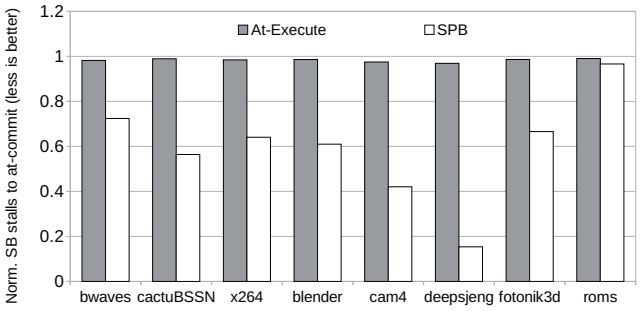
Fig. 8: Normalized SB stalls to at-commit for several SB sizes (less is better)

28 and 56 entries are 6.7%, 3.4% and 1.5% respectively. The benefits reach 16.8%, 9% and 4.3% respectively for SB-bound applications. This shows that SPB is not only faster than at-commit and at-execute strategies, but also saves energy (both static and dynamic). At-Execute (Exe. in the Figure) barely has any impact on energy consumption, with savings around 1%.

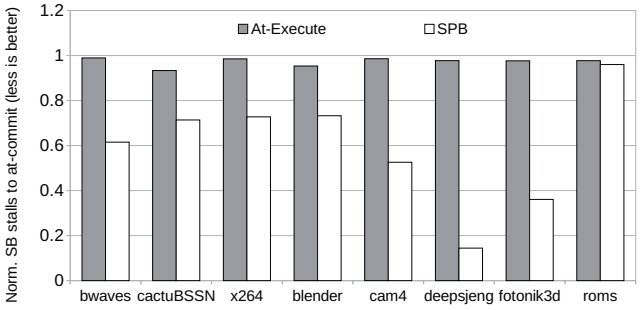
B. Impact on SB-induced stalls

The next step is to show the reduction of SB-induced stalls on the processor. Figure 8 shows the normalized SB stalls to at-commit strategy for different SB sizes. SPB manages to drop the average SB related stalls by 24% (worst, SB56) to 37% (best, SB28). However, we are still far from completely removing SB-induced stalls. Most of the remainder SB-induced stalls are either ”cold” stalls (before SPB detects the pattern), late prefetches (to be discussed later), or code sections that do not follow a pattern we can capture. Figure 9 shows the normalized stalls for SB-bound applications.

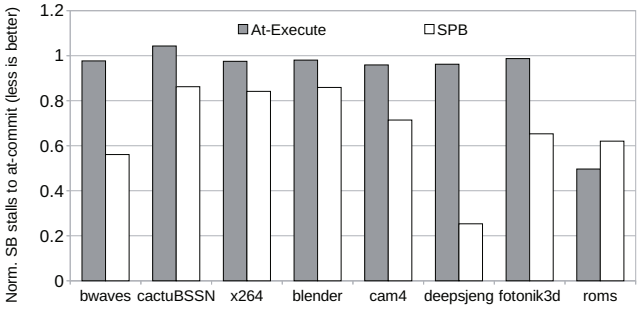
Looking at performance numbers in the previous section, we can observe that removing additional SB related stalls (Ideal SB) would only move the bottleneck to a different resource. Indeed, Figure 10 shows the normalized issue stalls to at-commit for all the studied SB sizes. The normalized values are broken down based on the source of the stalls, that can be either the SB, or another resource (Other) (e.g., reservation



(a) 14-entry SB

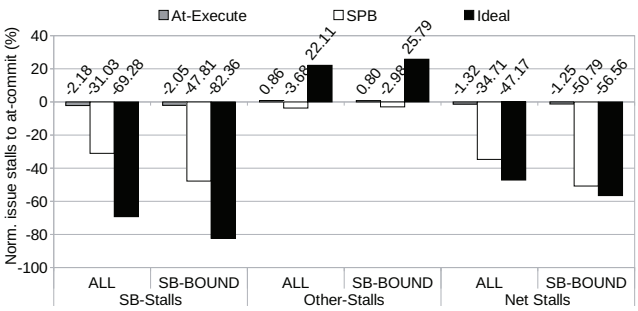


(b) 28-entry SB

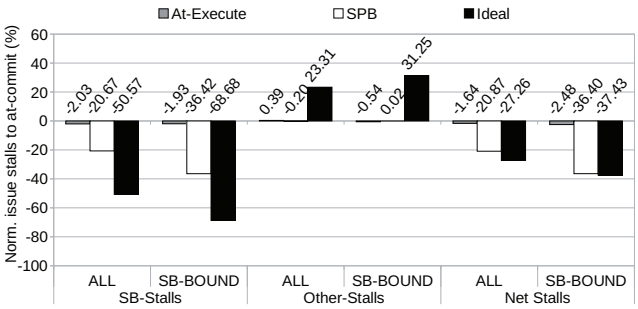


(c) 56-entry SB

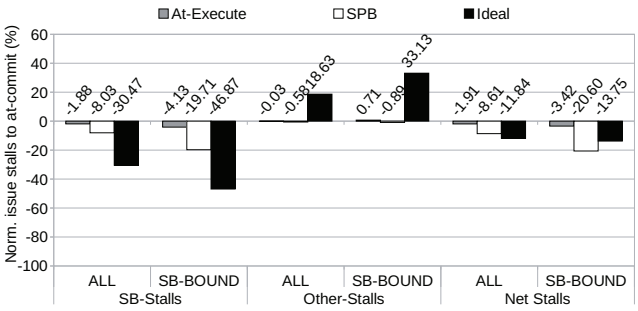
Fig. 9: Per SB-Bound application normalized SB stalls to at-commit for different SB sizes



(a) 14-entry SB



(b) 28-entry SB



(c) 56-entry SB

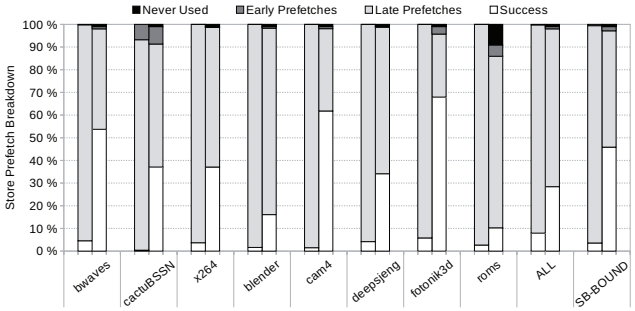
Fig. 10: Normalized issue stalls to at-commit for different SB sizes. X axis breaks down the source of the stalls, as well as shows the net stall reduction.

stations, reorder buffer, load queue, registers, etc). For example, for a SB of 14 entries (Figure 10-a), and considering the geometric mean of all SPEC CPU 2017, the ideal SB manages to reduce the total issue stalls by 69.3% compared to at-commit. The source of that reduction comes solely from SB stalls. At the same time, the ideal SB increases the total issue stalls by 22.1% because of lack of other resources. This leads to a net stall reduction of 47.2% at issue stage. On the same scenario, SPB is able to reduce 31% issue stalls related to the SB, while also decreasing issue stalls due to lack of other resources by 3.7%. This is related to the super-linear speedup of some applications (e.g., fotonik3d), since store prefetches reduce the wait times of future loads that would otherwise miss in L1, thus releasing "Other" resources faster than the ideal SB. This leads to a net stall reduction of 34.7%, 12.5% below the Ideal SB. Figures 10-b and 10-c also show net savings very close to the ideal SB, even outperforming this approach for SB-bound applications when using a SB of 56-entries.

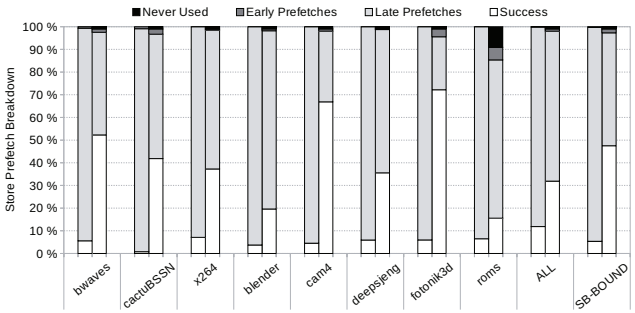
C. Accuracy and overheads

We have described how SPB affects performance and issue stalls. The next step is to break down how it behaves as a prefetcher compared to our default strategy (at-commit).

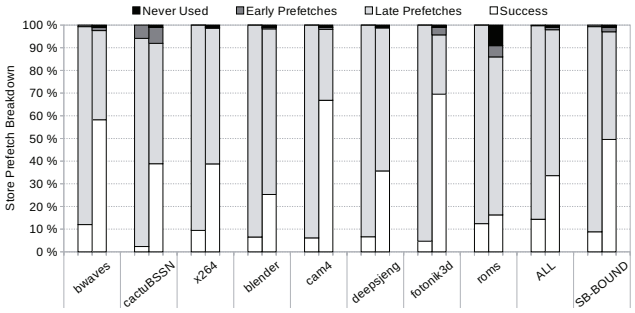
Figure 11 shows successful, late and early prefetches (either from invalidation or replacement), and never used blocks. For all applications SPB outperforms at-commit strategy in terms of accuracy. Success rate reaches 45% to 50% on SB-bound applications, while when considering all applications in the SPEC CPU 2017 suite, the success rate drops to around 30%. This is much higher than the success rates of at-commit, that ranges between 5 and 10%. Indeed, most prefetches performed



(a) 14-entry SB



(b) 28-entry SB



(c) 56-entry SB

Fig. 11: Breakdown of store requests at L1D level (first bar for at-commit, second for SPB).

by at-commit are late prefetches, since the prefetch request is generated at the end of the store's live cycle. On the other hand, SPB prefetches blocks much earlier assuming a sequential access pattern, increasing success rate (and also early prefetches by 2.5%). Note that this Figure considers all stores in the application, not only those that stall the pipeline.

It is also interesting to consider the impact on network traffic that SPB generates. A possible drawback of SPB would be the extra traffic generated by "false positive" prefetch requests. Our evaluation showed that over 97% of the prefetched bytes in each store burst are completely written by the application, while around 2% are almost completely written (98% of the bytes prefetched in the burst are actually written).

Figure 12 shows normalized prefetch block requests increment (to at-commit) from the CPU to the L1 controller (REQ). All these requests will check the tags to find a matching block.

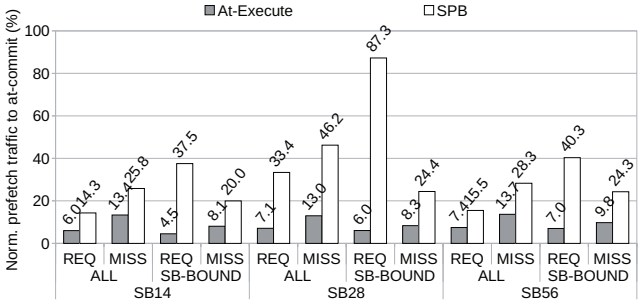


Fig. 12: Normalized prefetch traffic. Requests sent by the CPU (REQ) and blocks requested to the L2 (MISS)

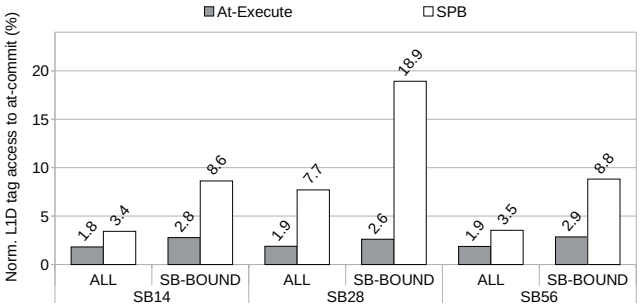


Fig. 13: Normalized L1 overhead (TAG accesses in L1D)

However, only those that miss will generate a L2 request and network traffic along the memory hierarchy (MISS). In relative terms (accounting prefetch accesses to the L1D cache), SPB generates additional 3.4% tag checks compared to at-commit for a SB of 14 entries, 7.7% for a SB of 28 entries and 3.5% for a SB of 56 entries (Figure 13). Prefetch traffic is higher for SB-bound applications, since SPB is enabled more often. In this scenario, prefetch traffic increases by 8.6%, 18.9% and 8.8% for SB sizes of 14, 28 and 56 entries respectively. However, since we are significantly reducing the amount of load requests to the L1D cache from the wrong path of execution, the overall access count to the L1D is reduced. Indeed, for store buffers of sizes 14 and 28, the average L1D access reduction is close to 1% for SPB. For a 56-entry store buffer, the net L1D access reduction reaches 2%. This, as we have seen, decreases slightly the power used by the L1D cache.

In order to show if SPB has any negative impact on L1D stalls due to additional network traffic we will rely on Intel's Top-Down model information. More specifically, in the metric execution stalls while there are L1D misses pending. Figure 14 shows how SPB additional traffic not only does not affect performance for most applications, but it also has a positive effect on the total amount of execution cycles the processor is stalled while waiting for L1D misses to be attended. Indeed, for a SB of 14 entries SPB reduces execution stalls by 27.2% compared to at-commit for all SPEC CPU 2017. When focusing on SB-Bound applications, the difference is significantly higher, reaching 52.8%. For a SB of 28 entries

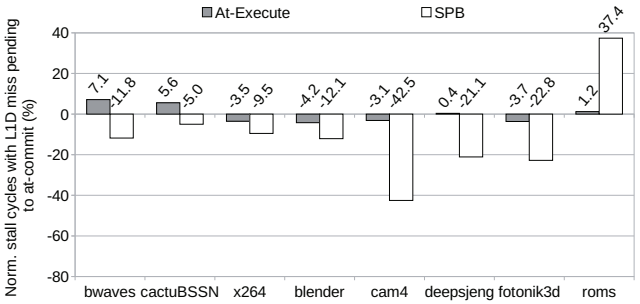


Fig. 14: Execution stalls with L1D misses pending

execution stalls are reduced by 12.2% and 30.4% respectively, while for a SB of 56 entries we reach some reasonable 3.9% and 12.6% reduction on execution stalls while there are L1D misses pending to be attended.

Figure 15 shows the per SB-bound application breakdown of execution stalls with L1D cache misses pending. As expected from the performance results, all the SB-bound applications except roms benefit from SPB. This is due to an artifact/pathology caused by SPB, that forces useful memory blocks out of the L1D in roms. More specifically, conflict misses increase by 10.3% for roms compared to the 1.2% on average for all SPEC CPU 2017 for a SB of 28 entries (10.6% to 1.9% for a SB of 56 entries).

D. Aggressive cache prefetchers

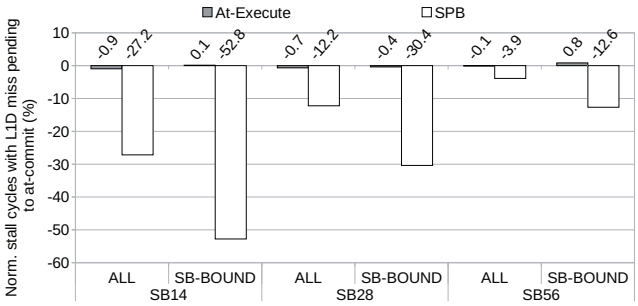
SPB is a highly selective and aggressive prefetching mechanism tailored to reduce SB-induced stalls. SPB requires little hardware modifications compared to other aggressive or adaptive prefetching schemes, such as the ones presented by Srinath *et al.* [28]. Moreover, these prefetching mechanisms apply load strategies blindly to stores.

This section compares SPB to sophisticated prefetching techniques presented in the literature [28]. For the technique in [28], we implement both the aggressive scheme and the adaptive prefetcher with the specified thresholds and modes.

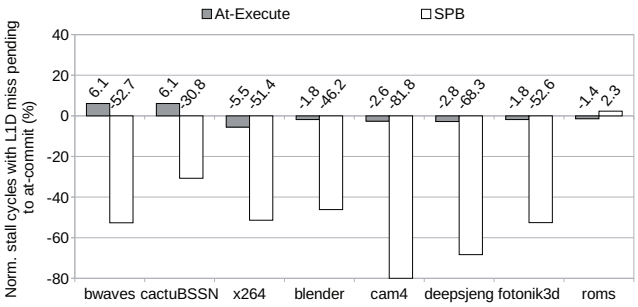
These cache prefetchers obtain better performance than our baseline stream (stride) prefetcher. Our goal is, however, to show that the use of SPB is still necessary to reduce SB-induced stalls even on top of aggressive cache prefetchers.

Figure 16 normalizes the performance to our ideal SB with the same generic prefetcher (stream, aggressive or adaptive). This way, we can see how far is each configuration from the ideal case, for each prefetcher and for at-commit and SPB.

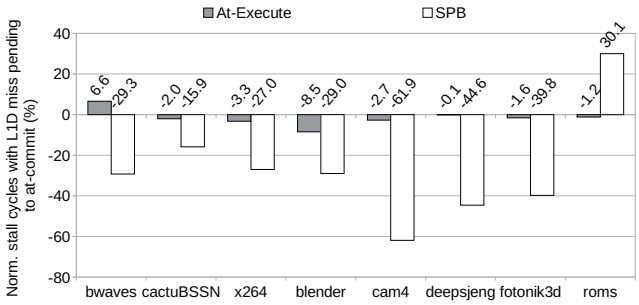
The aggressive and adaptive prefetchers do not have much impact in reducing the SB-induced stalls. These prefetching schemes still suffer from the same limitation as the stream prefetcher: prefetching requests are limited to those generated by the stores in the SB. The only difference is that they "shift" the prefetching window depending on how aggressive they are. On the other hand, SPB prefetches blocks for all addresses within the currently accessed memory page, going beyond the scope of the SB and performing an aggressive, yet controlled, prefetching only when a store burst is detected.



(a) 14-entry SB



(b) 28-entry SB



(c) 56-entry SB

Fig. 15: Per SB-Bound application normalized (to at-commit) execution stalls with L1D misses pending.

TABLE II: Configurations for the sensitivity analysis

Name	ROB	IQ	LQ	SQ	Width
SLM	32	15	10	16	4
NHL	128	32	48	36	4
HSW	192	60	72	42	8
SKL	224	97	72	56	8
SNC	352	128	128	72	8

E. SPB and core aggressiveness

This section discusses the effects of SPB on different core configurations, from simple and energy-efficient to complex high-performance cores. For this evaluation we consider 5 core configurations: Silvermont (SLM), Nehalem (NHL), Haswell (HSW), Skylake (SKL) and Sunny Cove (SNC). Sizes for the reorder buffer (ROB), the issue queue (IQ), the load queue (LQ), the store queue (SQ) and the back-end per-stage width (width) are shown in Table II.

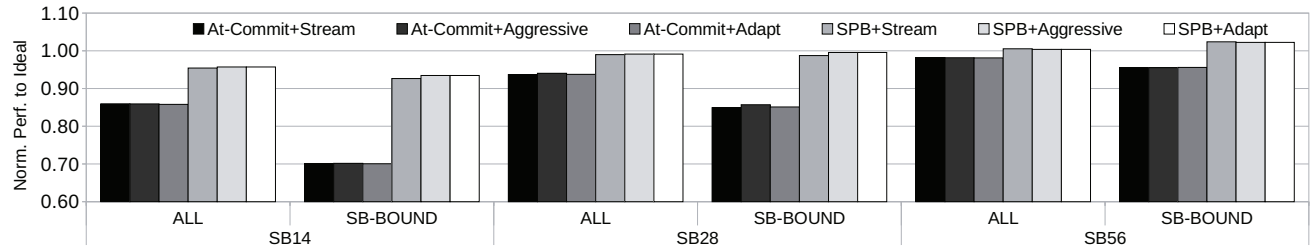


Fig. 16: Normalized execution time to Ideal + Prefetcher (stream, aggressive, adaptive) respectively

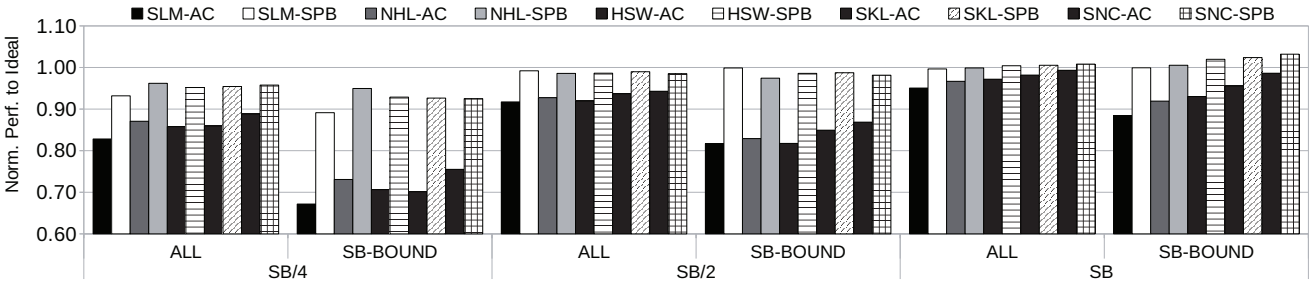


Fig. 17: Normalized execution time to Ideal for different core configurations

Figure 17 shows the normalized execution time (to an ideal SB), for the *at-commit* and SPB prefetch strategies. Results show that the performance gap between the ideal SB and the *at-commit* strategy increases as we move to energy-efficient core designs. However, SPB maintains ideal performance levels regardless of the architecture for the default SB size, and near ideal levels for a SB of half the size. Interestingly, SPB slightly outperforms the ideal SB for SNC, SKL and HSW-like configurations. As we showed in Section VI-C, this is because SPB reduces the waiting time of L1D misses (and that includes loads). SPB, as opposed to the ideal SB, aggressively prefetches memory blocks that are speculative, while both *at-commit* and ideal only prefetch memory blocks as stores commit. For reduced SB sizes, SPB delivers at least 89% of the ideal SB performance, while for that same configuration *at-commit* only manages to yield 67% of the ideal.

F. SPB in parallel applications

This section evaluates SPB for the parallel applications of the PARSEC benchmark suite for the purpose of showing that (i) multi-threaded applications also contain store bursts and (ii) SPB does not introduce a negative coherence effect.

Figure 18 shows the average performance both for all and for just the SB-bound PARSEC applications (i.e., *bodytrack*, *dedup*, *ferret*, and *x264*). As it happens for the sequential applications, SPB outperforms *at-commit* by 1% on average, and 1.1% if we only consider SB-bound applications. For a reduced SB size of 14 entries, SPB achieves 18.5% improvements for SB-bound applications, and around 4.3% on average for all benchmarks. There is no benchmark that suffers from performance degradation compared to the baseline for SPB, which shows that SPB is coherence-friendly. The reason is that (i) SPB is only enabled on a store burst scenario and (ii) store bursts do not happen for contended memory addresses, at least for the analyzed PARSEC benchmarks and inputs. Moreover, SPB does not produce cache pollution (at least for 8 threads), that would otherwise slow down performance.

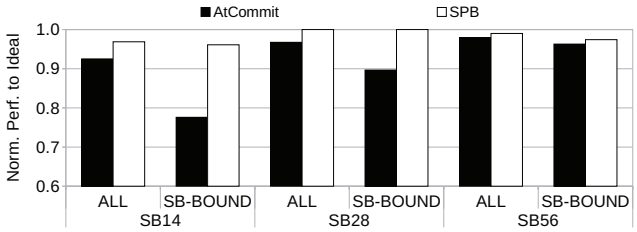


Fig. 18: Normalized execution time to Ideal for PARSEC benckmarks with 8 threads

VII. Related Work

A. Aggressive prefetchers

Cache prefetchers are usually applied over addresses that have been requested by either loads or stores. This means that they help to reduce issue stalls caused by both the filling of the load buffer or the SB. These prefetchers commonly employ a throttling mechanism to adjust the aggressiveness (degree) of the data prefetcher dynamically. Some prefetchers opt for being aggressive by default and reduce aggressiveness on cache pollution [28]. Other prefetchers on the other hand start with a low prefetching degree and seek for opportunities to increase aggressiveness over time [14]. SPB is always aggressive but highly selective, as it is specialized on a particular access pattern (stores to contiguous blocks), this it is less frequently triggered than generic cache prefetchers.

Spatial prefetchers are aggressive by default [7], [8], [27]. They collect the accessed blocks within a page and prefetch them again on the first access to that page. On the other hand, SPB targets stores to contiguous blocks that may happen due to a memory copy or a memory initialization. This may happen only once in the execution of a program, so learning the page is not an effective mechanism for SPB. Additionally the size of our prefetcher is minimal as it does not requires the tracking of previously accessed pages.

An alternative to prefetching blocks closer to the core is to send all data blocks directly to the last-level cache. That is the case of the store miss accelerator (SMA) [9]. In this proposal, the last-level cache serves as a buffer to stores that have an off-chip miss. When the associated block arrives to the last-level cache, the store data is combined with it. In contrast, we achieve virtually the same benefits (with respect to SLP) as SMA with a far simpler and cost-effective approach.

B. Reducing SB-induced stalls

There are other alternatives for reducing SB-induced stalls. The scalable store buffer (SSB) [31] eliminates stalls in the SB by maintaining stores in a large FIFO structure (1K entries). These stores write directly in the L2 cache. However, this alternative requires many hardware changes, including the coherence protocol, to address the invalidation of modified blocks in a cache.

Another alternative is to detect stores to blocks at compile time that can safely be performed out of order [26]. However, in the case of prefetch burst, all stores will miss in the L1 cache (and probably L2 and L3 too), and little benefit can be obtained from reordering. Finally, coalescing stores [24] can reduce the number of entries occupied by stores in the SB. Coalescing up to a memory block size, however, would entail to increase the size of the SB significantly. In contrast, we obtain performance figures very close to the ideal with minimal hardware overhead.

VIII. Conclusions

The size of the store buffer has been a critical factor for performance. Proof of this is the increment in SB size in Intel processors (from 32 to 56 entries) in just 10 years. Store prefetching is a fundamental technique for enabling MLP for stores. However, this MLP is restricted to the stores that fit in the store buffer, corresponding to just a few memory blocks.

Store Prefetch Burst (SPB) improves MLP outside the code scope delimited by the SB size. It detects the few store instructions that are responsible from most SB stalls, which are those that access contiguous memory blocks as a consequence of a memory copy or initialization. Then, it triggers an highly selective but aggressive prefetch request to the L1 controller that asks for all the remaining memory blocks in the current page. SPB is able to remove practically all SB-induced stalls and it is orthogonal to other cache prefetching strategies, proving to be a good addition for improved accuracy with a memory overhead of just 67 bits.

With SPB, a 56-entry SB reaches 100.5% of an ideal SB, and a 20-entry SB achieves the performance of a 56-entry SB with an at-commit prefetching policy. SPB excels for limited SB sizes. In a 14-entry SB scenario, prefetching at commit only achieves 85.9% of the ideal performance, while SPB achieves 95.4%, on average. SPB improvements are not limited to performance, but also apply to energy efficiency as it can achieve energy savings of 6.7% for a 14-entry SB, reaching 16.8% for SB-bound applications.

References

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[2] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 399–411.

[3] R. Bera, A. V. Nori, , O. Mutlu, and S. Subramoney, "Dspatch: Dual spatial pattern prefetcher," in *52nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2019.

[4] E. Bhatia, G. Chacon, S. H. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 1–13.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.

[7] J. F. Cantin, M. H. Lipasti, and J. E. Smith, "Stealth prefetching," in *12th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Oct. 2006, pp. 274–282.

[8] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos, "Accurate and complexity- effective spatial pattern prediction," in *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2004, pp. 276–287.

[9] Y. Chou, L. Spracklen, and S. G. Abraham, "Store memory-level parallelism optimizations for commercial applications," in *38th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Nov. 2005, pp. 183–196.

[10] A. Fog, "Instruction Tables. Instruction latencies, throughputs and micro-operation breakdowns," 2018, Available at http://www.agner.org/optimize/instruction_tables.pdf.

[11] A. Gandhi, H. Akkary, R. Rajwar, S. T. Srinivasan, and K. Lai, "Scalable load and store processing in latency-tolerant processors," *IEEE Micro*, vol. 26, no. 1, pp. 30–39, Jan 2006.

[12] K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance evaluation of memory consistency models for shared-memory multiprocessors," *SIGPLAN Not.*, vol. 26, no. 4, pp. 245–257, Apr. 1991.

[13] ——, "Two techniques to enhance the performance of memory consistency models," in *20th Int'l Conf. on Parallel Processing (ICPP)*, Aug. 1991, pp. 355–364.

[14] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, "Near-side prefetch throttling: Adaptive prefetching for high-performance many-core processors," in *27th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Nov. 2018, pp. 28:1–28:11.

[15] Intel, "Intel® 64 and ia-32 architectures optimization reference manual," www.intel.com, 2019.

[16] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *49th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Oct. 2016, pp. 60:1–60:12.

[17] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," in *22nd Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Apr. 2017, pp. 737–749.

[18] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2009, pp. 469–480.

[19] P. Michaud, "Best-offset hardware prefetching," in *22nd Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Mar. 2016, pp. 469–480.

[20] D. Molka, R. Schöne, D. Hackenberg, and W. E. Nagel, "Detecting memory-boundedness with hardware performance counters," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 27–38.

[21] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. fei Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2014, pp. 626–637.

[22] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso, "Performance of database workloads on shared-memory systems with out-of-order processors," *SIGPLAN Not.*, vol. 33, no. 11, pp. 307–318, Oct. 1998.

[23] P. Ranganathan, V. S. Pai, and S. V. Adve, "Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models," in *Symp. on Parallel Algorithms and Architectures (SPAA)*, 1997, pp. 199–210.

[24] A. Ros and S. Kaxiras, "Non-speculative store coalescing in total store order," in *45th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2018, pp. 221–234.

[25] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.

[26] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *39th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2012, pp. 524–535.

[27] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *33rd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2006, pp. 252–263.

[28] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *13th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2007, pp. 63–74.

[29] Thin-Fong Tsuei and W. Yamamoto, "Queuing simulation model for multiprocessor systems," *IEEE Computer*, vol. 36, no. 2, pp. 58–64, Feb 2003.

[30] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "TriCheck: Memory model verification at the trisection of software, hardware, and ISA," in *22nd Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Apr. 2017, pp. 119–133.

[31] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," in *34th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2007, pp. 266–277.

[32] S. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in McPAT and potential impacts on architectural studies," in *21st Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2015, pp. 577–589.

[33] A. Yasin, "A top-down method for performance analysis and counters architecture," *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pp. 35–44, 2014.