Exploiting Vector Code Semantics for Efficient Data Cache Prefetching

Francesc Martínez Palau
Barcelona Supercomputing Center
Barcelona, Spain
francesc.martinez@bsc.es

Martí Torrents
Barcelona Supercomputing Center
Barcelona, Spain
marti.torrents@bsc.es

Adrià Armejach
Barcelona Supercomputing Center
Barcelona, Spain
Universitat Politecnica de Catalunya
Barcelona, Spain
adria.armejach@bsc.es

Marc Casas
Barcelona Supercomputing Center
Barcelona, Spain
marc.casas@bsc.es

ABSTRACT

Emerging workloads from domains like high performance computing, data analytics or deep learning consume large amounts of memory bandwidth. To mitigate this problem, computing systems include large and deep memory cache hierarchies that exploit both spatial and temporal locality. In this context, hardware data cache prefetching constitutes a useful method to anticipate cache misses and boost performance. Despite their success in terms of high coverage rates, current data cache prefetchers incur a significant number of late and sometimes useless prefetches. Additionally, these state-of-the-art prefetchers are not aware of architecture trends towards larger vector units and vector-length agnostic instruction sets.

This paper demonstrates that these trends bring new prefetching opportunities that make it possible to increase the accuracy and timeliness of any state-of-the-art prefetcher with a negligible area cost. We propose the the *Register Vector Length Agnostic* (*ReVeLA*) prefetcher. ReVeLA exploits program semantics present in vectorized codes. The ReVeLA prefetcher complements existing data cache prefetchers by providing highly accurate prefetch requests that improve prefetching timeliness and accuracy without significantly increasing memory bandwidth consumption. When applied on top of a state-of-the-art out-of-order vector processor, ReVeLA delivers a speed-up of 1.23× with respect to a system without any prefetching approach. When combined with the $NextLine$, *BOP*, *SPP*, and *PPF* prefetchers, ReVeLA improves performance by 6.57%, 4.46%, 11.83%, and 11.40% respectively, with respect to a vector processor equipped with these prefetching approaches. Additionally, our evaluation demonstrates that ReVeLA increases memory bandwidth consumption by only 3.74% when combined with the most performing data cache prefetcher of our experimental campaign.

ACM Reference Format:
Francesc Martínez Palau, Martí Torrents, Adrià Armejach, and Marc Casas. 2024. Exploiting Vector Code Semantics for Efficient Data Cache Prefetching. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650200.3656635

1 INTRODUCTION

Workloads from domains like high performance computing, data analytics, or deep learning consume large amounts of memory bandwidth to feed power-hungry hardware compute units. This issue, described as the *Memory Wall* [25, 44], has become worse in recent years. To mitigate it, computing systems contain large and deep memory cache hierarchies exploiting the spatial and temporal locality that some workloads exhibit. In this context, hardware cache prefetching significantly contributes to tackle the Memory Wall. State-of-the-art cache prefetchers have pushed the limits of data prefetching by exploiting recurring spatial patterns of physical addresses [4, 6, 16, 21, 26, 29, 32]. However, these prefetching approaches face the classical trade-off between accuracy and coverage. The more accurate they are, the less coverage they provide, and vice versa. Since reducing data cache misses is crucial, computer architects and hardware vendors have designed and implemented prefetchers with large coverage that spend a significant amount of memory bandwidth bringing memory blocks to the cache hierarchy. These prefetched blocks are frequently late or sometimes not even consumed by the running workloads. Furthermore, these state-of-the-art prefetchers do not exploit recent trends in computer architecture, leaving performance on the table. For example, the recent re-emergence of vector architectures [28], which includes new vector Instruction Set Architectures (ISAs) like the Scalable Vector Extension (SVE) [34] and the RISC-V Vector Extension [36], and production systems equipped with long vector units [22, 35].

These emerging vector ISAs support Vector Length Agnostic (VLA) programming models [34], allowing parallel codes to run in a vector processor regardless of the size of its vector registers. In contrast, previously existing SIMD ISAs, like AVX-512 [31], are tied to a certain vector length that cannot be changed. This paper demonstrates that VLA vector ISAs bring new prefetching opportunities that make it possible to increase the accuracy and timeliness

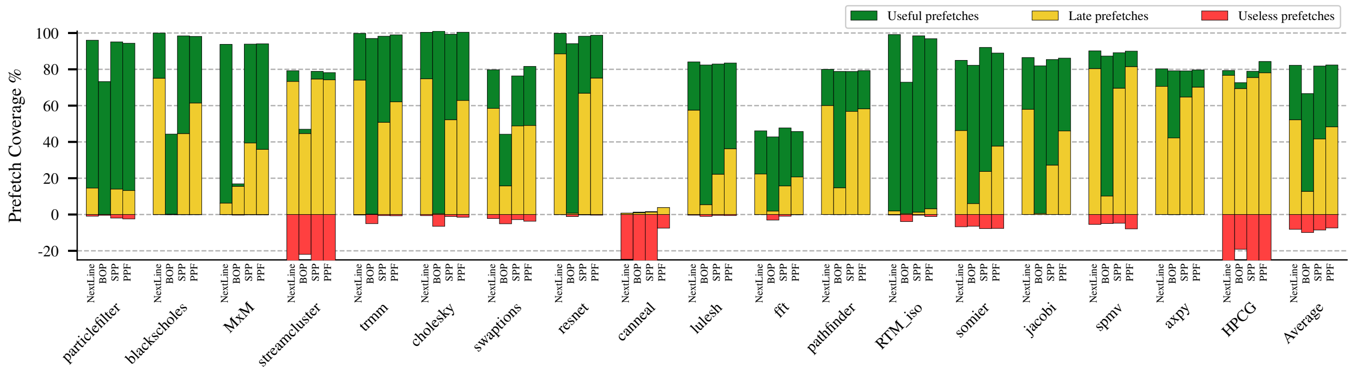


Figure 1: Prefetcher coverage of the different configurations.

of any state-of-the-art prefetcher with a negligible area cost. We propose a novel prefetching method called the *Register Vector Length Agnostic* prefetcher (*ReVeLA*). ReVeLA leverages program semantics present in vectorized codes to identify data streams that will be accesses with certainty. That is, VLA codes typically request access to data streams (*i.e*, *requested vector length*), which are larger than the implemented register vector length (*i.e.*, *granted vector length*) of the vector processor where they run. This enables opportunities to prefetch the data corresponding to the whole stream, which are known in advance. For example, if a vectorized code asks for a vector length of 2KB to load 256 64-bit elements while running on a vector processor that only supports 8 64-bit element vector instructions, it will be able to load only 8 elements in a single vector load and will have to load the remaining 248 elements during subsequent iterations. In this context, prefetching some of these 248 remaining elements before the corresponding vector loads request them significantly improves performance.

The ReVeLA prefetcher complements existing cache prefetchers by providing highly accurate prefetch requests that improve prefetching timeliness and accuracy without significantly increasing memory bandwidth consumption. ReVeLA only requires a small table to keep track of the already existing access data streams, very simple control logic to orchestrate the prefetch requests, and a small prefetch queue to keep track of pending prefetch requests. A very significant difference between ReVeLA and state-of-the-art data cache prefetchers is that ReVeLA does not trigger prefetch requests once a cache miss has taken place. Instead, ReVeLA checks its hardware structures every cycle and only triggers prefetch requests as long as the memory subsystem is not overloaded. This approach makes it possible to issue very accurate prefetches without wasting valuable memory bandwidth.

This paper makes the following contributions beyond the state-of-the-art:

- It proposes *ReVeLA*, the first prefetching mechanism that leverages program semantics of vectorized codes regarding the vector length. ReVeLA complements existing prefetching approaches by triggering highly accurate prefetch requests when the memory subsystem is not overloaded. ReVeLA incurs a minimal storage overhead of just 436 Bytes, and does not require any specific code modifications. The ReVeLA approach is supported by relevant vector ISAs like SVE [34] or RISC-V Vector [36].
- It proposes combining ReVeLA with already existing data cache prefetchers to improve their accuracy without hurting coverage, while incurring minimal hardware and memory bandwidth cost.
- It evaluates the impact of ReVeLA on 18 workloads that represent a variety of application domains, arithmetic intensities, and memory access patterns. When applied on top of a state-of-the-art out-of-order vector processor, ReVeLA delivers a speed-up of 1.23× with respect to a system without any prefetching approach. When combined with the *NextLine* [30], *BOP* [26], *SPP* [21], and *PPF* [7] prefetchers, ReVeLA improves performance by 6.57%, 4.46%, 11.83%, and 11.40%, respectively, with respect to a vector processor equipped with these prefetching approaches. Additionally, our evaluation demonstrates that ReVeLA only increases memory bandwidth consumption by 3.74% when applied on top of the most performing data cache prefetcher that we consider, while improving the number of useful prefetches by 11.09%.

Section 2 introduces the background and motivates our proposal, Section 3 introduces the ReVeLA prefetcher, Section 4 explains the evaluation methodology, Section 5 details the obtained results, Section 6 summarizes related work, and Section 7 describes the final the conclusions.

2 BACKGROUND AND MOTIVATION

2.1 Limitations of Data Cache Prefetching

Hardware data cache prefetching has achieved very large coverage rates by exploiting spatial patterns between past data cache misses and extrapolating these patterns to future accesses [4, 6, 16, 21, 26, 29, 32]. These prefetching techniques have also been implemented in real products due to their relatively low area overhead [1, 11, 13, 31]. Despite this success, data cache prefetching approaches still suffer from a large number of late prefetches that, if triggered before, could potentially boost the performance of the running workloads. Also, current prefetching approaches trigger a non negligible number of inaccurate prefetches that consume memory bandwidth and pollute caches. To illustrate these issues, we consider three state-of-the art spatial prefetchers for lower level

```
 1 void axpy(double a, double *x, double *y, int n) {
 2  int gvl;
 3  for (int i = 0; i < n; i += gvl) {
 4   gvl = riscv_vsetvl_e64m1(n-i);
 5   vfloat64m1_t vx = riscv_vle64_v_f64m1(&x[i], gvl);
 6   vfloat64m1_t vy = riscv_vle64_v_f64m1(&y[i], gvl);
 7   vy = riscv_vfmacc_vf_f64m1(vy, a, vx, gvl);
 8   riscv_vse64_v_f64m1(&y[i], vy, gvl);
 9  }
10 }
```

Figure 2: RISC-V "V" *axpy* vector length agnostic code example.

caches: the Best Offset Prefetcher (BOP) [26] the Signature Path Prefetcher (SPP) [21], and the Perceptron-based Prefetch Filtering (PPF) prefetcher [21]. For completeness, we also take into account the simple NextLine prefetcher [30].

Figure 1 shows our evaluation in terms of coverage considering these three prefetchers. Section 4 describes our experimental setup including the workloads we consider. In the x-axis we show all considered workloads, and per each workload we show four bars, one per considered prefetcher. In the y-axis we represent coverage in terms of percentage of Last Level Cache (LLC) misses that are served by prefetching requests. Our evaluation indicates that NextLine, BOP, SPP, and PPF reach 82.23%, 66.63%, 81.81%, and 82.38% average coverage rates, which are high numbers. However, the prefetches that arrive on time to serve misses (*useful* category) cover only 30.05%, 53.93% 40.20%, and 34.09% of the total misses for NextLine, BOP, SPP, and PPF respectively. Additionally, these prefetches trigger a non-negligible amount of *useless* prefetches, i.e., they consume bandwidth to bring memory blocks that are never accessed by the running workloads. This paper proposes a prefetching solution to increase timeliness and accuracy of current prefetching approaches by exploiting program semantics present in codes running on processors supporting VLA ISAs.

2.2 Vector Processors and Vector Length Agnostic ISAs

Recent years have seen a renewed interest in vector architectures, especially those featuring long vector registers, as evidence by production systems equipped with long vector units [22, 35]. In addition, emerging VLA ISAs such as RISC-V Vector [36] and SVE [34], make it possible to write semantically reach vector codes that directly expose information about future data access patterns to the hardware.

For example, Figures 2 and 3 show two VLA codes implementing the *axpy* kernel that use RISC-V Vector and SVE intrinsic calls, respectively. In both examples the main loop initially calls an instruction to define the number elements in x and y arrays to be processed in the current iteration. In the case of the RISC-V Vector code, this definition is done via the *vsetvl* instruction, which sets the vector length *gvl* to be the minimum between the number of elements that fit in a vector register and $n - i$ (line 4 of Figure 2). Similarly, the SVE code uses the *svwhilelt* instruction to generate a predicate indicating the number of elements to process, which is also the minimum between the number of elements that fit in one vector register and $n - i$. After this definition, both codes proceed in an analogous way: while the RISC-V code uses the vector length *gvl* to indicate the number of elements of arrays x and y to be be

```
 1 void axpy(double a, double *x, double *y, int n) {
 2  svbool_t predicate;
 3  for (int i = 0; i < n; i+=svcntd()) {
 4   predicate = svwhilelt_b64(i, n);
 5   svfloat64_t vx = svld1(predicate, &x[i]);
 6   svfloat64_t vy = svld1(predicate, &y[i]);
 7   vy = svmla_x(predicate, vy, vx, a);
 8   svst1(predicate, &y[i], vy);
 9  }
10 }
```

Figure 3: SVE *axpy* vector length agnostic code example.

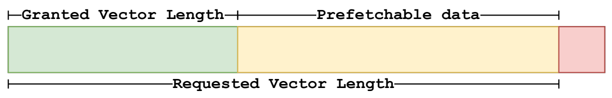


Figure 4: Example of the information available on a Requested Vector Length Agnostic system.

loaded by instructions *vle*64 (lines 5 and 6 of Figure 2), the SVE code does so by applying the predicate to instructions *svld*1. Similarly, these two codes execute vector Fused-Multiply and Add (FMA) and store instructions using either the vector length or the predicate. The codes displayed in Figures 2 and 3 run independently of the maximum vector length supported by the hardware, *i.e.*, they are VLA codes.

2.3 Motivation of ReVeLA

The ReVeLA prefetcher is motivated by the observation that information exposed to the hardware by VLA codes can be leveraged to accurately predict future memory accesses. ReVeLa exploits the difference between the intended amount of data to be processed by vector codes, and the maximum amount of data that a single vector instruction can handle. The most relevant vector ISAs, RISC-V Vector and SVE, support this approach. In the case of RISC-V, the ISA specification [36] makes it possible for a program to specify the size of the data to be processed in a certain operation. We call this size *Requested Vector Length (RVL)*. The processor grants the minimum value between this RVL and the maximum vector length it supports. We call this value provided by the processor *Granted Vector Length (GVL)*. Figure 2 shows in line 4 how the *vsetvl* instruction returns the GVL value in the case of the RISC-V Vector code. In this scenario, the RVL value $n - i$ is an input of the *vsetvl* instruction. In the case of SVE, the *svwhile** instructions expose to the hardware the intended amount of data to be processed, *i.e.*, the RVL as Figure 3 indicates. These instructions generate predicates describing the number of elements to process. The number of active elements in the predicate corresponds to the minimum value between this RVL and the maximum vector length supported by the hardware, *i.e.*, the GVL. For the rest of the paper we use the RVL and GVL concepts without lose of generality since they are applicable to both RISC-V Vector and SVE ISAs.

ReVeLA exploits the difference between RVL and GVL to predict future data accesses and drive data prefetching. Figure 4 displays a RVL value corresponding to a large and contiguous memory block, and a smaller GVL value corresponding to the maximum amount of data that vector instructions can process. While in the current iteration the hardware only loads the contiguous memory block corresponding to the first GVL elements, the larger RVL value indicates that a larger block will be accessed in the future. Data belonging to this large block that are beyond the first GVL elements constitute good candidates for prefetching since they will be accessed in the future. While current data prefetching approaches ignore this information, ReVeLA uses is to generate prefetches. ReVeLA is not conceived to be a standalone prefetcher, but to work alongside state-of-the-art data prefetchers. ReVeLA incurs small area overhead and provides accurate prefetch requests that improve prefetching timeliness and accuracy without significantly increasing memory bandwidth consumption.

3 THE REVELA PREFETCHER

The ReVeLA prefetcher exploits streams of memory accesses for which the vector code provides the base address and total size in advance, *i.e.*, before triggering the memory accesses. The ReVeLA hardware components keep track of these streams to predict future memory accesses. To update the content of these components, ReVeLA takes into account data demand memory requests. The ReVeLA prefetch triggering logic is not associated with events like cache misses. Instead, ReVeLA takes into account the status of the prefetch queues and cache MSHRs to trigger prefetches, which avoids overwhelming the memory subsystem with prefetches when the number of in-flight operations is close to the maximum. Section 3.1 presents the design of the ReVeLA prefetcher, Section 3.2 discusses the update of the ReVeLA hardware components, Section 3.3 describes how ReVeLA triggers prefetches, Section 3.4 provides an example illustrating ReVeLA operation, Section 3.5 discusses how ReVeLA deals with memory address translation, and Section 3.6 demonstrates that the area overhead of ReVeLA is just 436 Bytes.

3.1 ReVeLA design

The ReVeLa design keeps track of vector accesses belonging to large memory streams. To do so, ReVeLA requires propagating the RVL and GVL values of memory instructions to each individual load/store request. These values make it possible for ReVeLa to identify the application memory access streams via the *Stream Tracking Table (STT)*. Figure 5 displays the STT design. Each entry of the STT tracks one access stream. The STT is indexed by the stream limit address, which is defined as the base stream address plus its corresponding RVL value ($limit@$ in Figure 5), as it remains constant during the whole stream. Each STT entry stores the last accessed memory address of its stream ($current@$ in Figure 5), the number of cache lines after $current@$ that have already been prefetched ($prefetched\_distance$), a valid bit, and an 8-bit LRU counter. ReVeLA's aggressiveness is throttled depending on the number of valid STT entries using a simple hardware structure, the *Aggressivity Table*. This table adapts the aggressiveness depending on the number of valid streams, *i.e.*, allowing more prefetch requests per stream if the stream count is low. Figure 5 shows an example. ReVeLA uses a small 16-entry *Prefetch Queue* to store the pending prefetch requests, which is a common practice in cache prefetching [15].

3.2 Update of the STT table

STT is updated for each new demand memory access requested by a vector instruction. As Figure 6 shows, ReVeLA computes the stream limit address (requested address + RVL) of the new memory request,

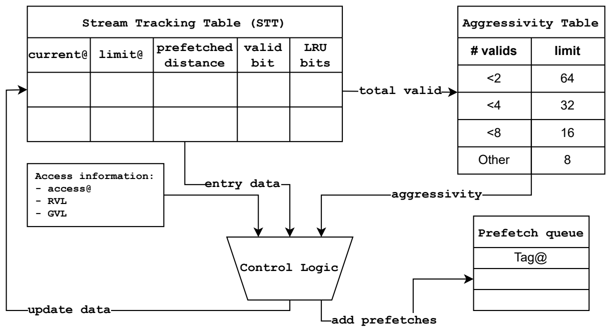


Figure 5: Architecture of ReVeLA's Stream Tracking Table (STT).

and uses this limit address to look up the SST table. If there is a hit on a valid STT entry and the hitting address (*access@*) is a forward access, meaning the address stored in the *current@* field is smaller or equal than the *access@*, ReVeLA updates the following fields of the entry: the *current@* field, which is updated with *access@* + GVL, the $prefetched\_distance$, which is updated to contain the number of cache lines after the new *current@* that have already been prefetched, and the LRU counter of the hitting entry is set to 0. In addition, the LRU counters of all the other STT entries are increased by 1. If there is a hit, but *access@* is smaller than the current address, it is considered an access to a past region of the stream, and no content on the STT is updated. The reason why hitting demand accesses with addresses smaller than *current@* do not trigger STT updates is to avoid storing outdated data due to out-of-order execution and other microarchitecture mechanisms that may reorder memory accesses belonging the same memory stream. Finally, if a hit to an entry is also the final access of the stream (*i.e. entry.current@ = entry.limit@*) the entry on the STT is invalidated. If there is a miss on the STT table, ReVeLA creates a new entry on STT overwriting either an invalid entry or, if all entries are valid, the one with the largest LRU counter.

To avoid stale entries, ReVeLA invalidates STT entries when their LRU counter reaches its maximum value, *i.e.*, 255 in the case of 8-bit counters. If we do not invalidate these stale entries after an inactivity period, they will remain in the STT and impact the prefetcher accuracy or the prefetcher aggressivity, which depends on the number of valid STT entries. There are several scenarios that may create stale entries. For example, if the last two accesses to a stream are executed out of order, the last stream access will invalidate the stream entry but when the second to last arrives, a new entry will be created. In this scenario, if the stream is accessed again from the beginning, the stale entry will prevent any prefetches to be issued again on that memory stream.

3.3 ReVeLa prefetch trigger

This section describes how ReVeLA triggers prefetch requests. The triggering logic of ReVeLa is not driven by events like cache misses or accesses. Instead, the prefetch triggering logic is evaluated every cycle and, as long as the memory system can accept requests from the prefetch queue and this prefetch queue is not full, ReVeLA inserts prefetch requests into the prefetch queue. First, ReVeLA

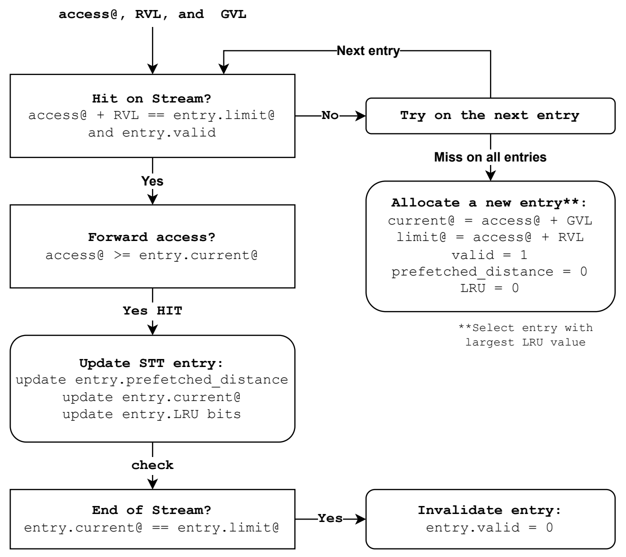


Figure 6: ReVeLA logic when processing memory accesses.

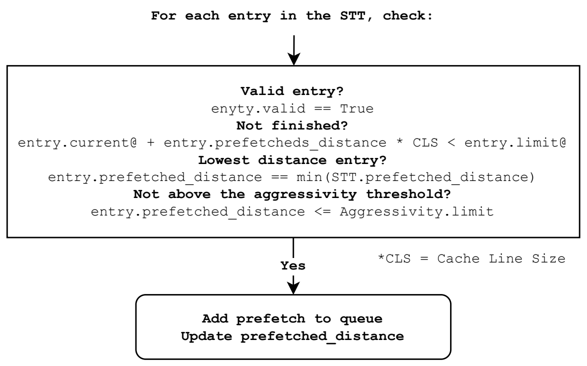
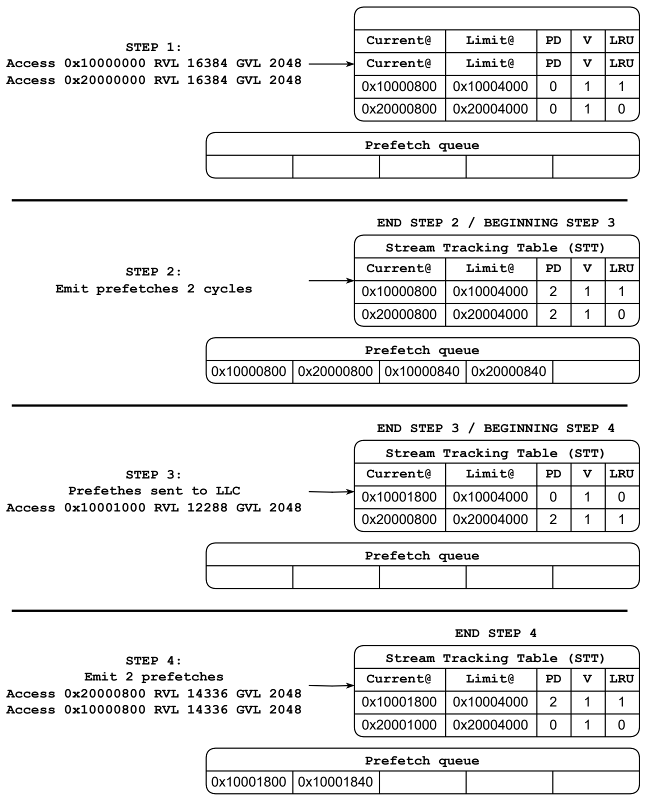

For each entry in the STT, check:
Valid entry?
enyty.valid == True
Not finished?
entry.current@ + entry.prefetcheds_distance * CLS < entry.limit@
Lowest distance entry?
entry.prefetched_distance == min(STT.prefetched_distance)
Not above the aggressivity threshold?
entry.prefetched_distance <= Aggressivity.limit
Yes
*CLS = Cache Line Size
Add prefetch to queue
Update prefetched_distance

Figure 7: ReVeLA logic when emitting prefetches.


STEP 1:
Access 0x10000000 RVL 16384 GVL 2048
Access 0x20000000 RVL 16384 GVL 2048

Current@	Limit@	PD	V	LRU
Current@	Limit@	PD	V	LRU
0x10000800	0x10004000	0	1	1
0x20000800	0x20004000	0	1	0

Prefetch queue				

END STEP 2 / BEGINNING STEP 3
STEP 2:
Emit prefetches 2 cycles
Stream Tracking Table (STT)

Current@	Limit@	PD	V	LRU
0x10000800	0x10004000	2	1	1
0x20000800	0x20004000	2	1	0

Prefetch queue				
0x10000800	0x20000800	0x10000840	0x20000840	

END STEP 3 / BEGINNING STEP 4
STEP 3:
Prefethes sent to LLC
Access 0x10001000 RVL 12288 GVL 2048
Stream Tracking Table (STT)

Current@	Limit@	PD	V	LRU
0x10001800	0x10004000	0	1	0
0x20000800	0x20004000	2	1	1

Prefetch queue				

END STEP 4
STEP 4:
Emit 2 prefetches
Access 0x20000800 RVL 14336 GVL 2048
Access 0x10000800 RVL 14336 GVL 2048
Stream Tracking Table (STT)

Current@	Limit@	PD	V	LRU
0x10001800	0x10004000	2	1	1
0x20001000	0x20004000	0	1	0

Prefetch queue				
0x10001800	0x10001840			

Figure 8: ReVeLA operation step-by-step.

computes the number of STT valid entries and uses it to set a maximum prefetch distance. ReVeLa obtains this maximum prefetch distance from the aggressivity table by considering the number of valid entries present in the STT. Figure 7 illustrates how ReVeLa inserts new requests into the prefetch queue. For all valid STT entries, ReVeLA checks whether the base address (*entry.current@* in Figure 7) plus the prefetch distance (*entry.prefetched_distance*) is below the limit address (*entry.limit@*). For all entries with values below the limit address, ReVeLA considers the ones storing the minimum value of the $prefetched\_distance$ field considering all the STT valid entries. For all entries holding this minimum value, ReVeLA adds new prefetch requests to the queue if this minimum *entry.prefetched_distance* is below the maximum prefetch distance (*Aggressivity.limit* in Figure 7).

The number of new prefetch requests per entry is determined by the $prefetched\_distance$ parameter. The $prefetched\_distance$ field of all entries triggering prefetches is updated to account for these new prefetch requests inserted in the prefetch queue. The cache level where ReVeLA is operating checks whether there are pending prefetch requests in the ReVeLA prefetch queue every cycle. To avoid overwhelming the cache with too many prefetch requests, the cache does not insert prefetch requests in its Miss Status Holding Register (MSHR) if there are less than 8 free MSHR entries.

3.4 Example of ReVeLa Operation

To illustrate ReVeLA operation, Figure 8 shows an example to describe how ReVeLA updates STT entries once the CPU emits a sequence of demand memory requests, and how ReVeLA triggers prefetch requests. On Step 1, two different vector memory accesses are triggered by the CPU. We assume the STT table to be empty, so both vector accesses miss in the STT. Therefore, ReVeLA allocates two new STT entries, one for each access, with the corresponding *current@* (address + GVL), *limit@* (address + RVL), $prefetched\_distance$ (*PD*) (zero), valid bit and LRU counter. During Step 2, we assume that ReVeLA has two cycles without processing memory requests, so it executes the prefetch triggering logic two times. Since both entries have the lowest prefetched distance, both generate prefetch requests and add them to the prefetch queue. These requests are expressed in terms of 64 bytes cache blocks. On Step 3, ReVeLA triggers the prefetch requests to the memory subsystem, and processes a new demand vector memory request. This request hits in the entry with *limit@* 0x10004000 and, therefore, ReVeLA updates the *current@* and $prefetched\_distance$ fields of the corresponding STT entry. In this case, ReVeLA sets the prefetch distance to zero indicating that nothing has been prefetched from the *current@* to the *limit@*.

On Step 4 we assume ReVeLA has two cycles to emit prefetch requests. Since only the STT entry with *limit@* 0x10004000 contains the smallest prefetch distance, ReVeLA emits requests just for this entry. As Figure 8 indicates, ReVeLA emits two requests, 0x10001800 and 0x10001840, in the two cycle period. After the prefetch requests are emitted, ReVeLA has to process two new demand vector memory requests. The first one, with base address 0x2000800, hits in the STT entry with limit address 0x20004000. Therefore, ReVeLA updates the *current@*, prefetched distance, and LRU bits fields of the hitting entry. Then, ReVeLA processes the demand vector access with base address 0x1000800. This access maps to the STT entry with *limit@* 0x10004000. Since the base address is below *current@*, the STT entry remains unmodified as we describe in Section 3.2.

3.5 ReVeLA and Memory Address Translation

ReVeLA works with virtual addresses to ensure the address range contiguity of the streams being tracked by the STT. The use of physical addresses would complicate stream definitions on the STT as addresses may not be contiguous when crossing page boundaries. To be exposed to virtual addresses, ReVeLA hardware structures are placed alongside the first level caches and accessed before the address translation process is finished. ReVeLA prefetch requests follow the same path as demand memory requests triggered by vector instructions. Since all cache levels are physically tagged, ReVeLA prefetch requests must go through the Translation Lookaside Buffer (TLB) and potentially trigger TLB misses and subsequent page walks to complete cache lookups. Therefore, ReVeLA not only acts as a cache prefetcher, but it may also prefetch memory page translations into the TLB.

3.6 Area Cost of ReVeLA

The area cost of ReVeLA is minimal since each STT entry requires 64 bits for the *current@* field, 64 bits for *limit@*, 8 bits for the *PrefetchedDistance*, 8 bits to guide LRU replacement, and one bit to indicate whether the entry is valid or not. This adds up to 145 bits per entry. Our experimental campaign considers a 16-entry STT, which means the total STT size is 2320 bits or 290 Bytes. Section 5.4 shows that a 4-entry STT already achieves 91.3% of the potential ReVeLA performance benefits, which indicates that the area cost of STT can be reduced to just 73 Bytes and still provide very significant performance improvements. The prefetch queue requires 128 Bytes to store 16 entries. The cost of the aggressivity table is also minimal, with only 8 Bytes of information. The total storage overhead of ReVeLA is 436 Bytes when using a 16-entries STT, a 16-entries prefetch queue, and a small aggressivity table like the one that Figure 5 shows. ReVeLA also requires some extremely simple control logic that incurs a negligible overhead.

4 EXPERIMENTAL ENVIRONMENT

4.1 Simulation Methodology

Our simulation methodology is based on ChampSim [2], a detailed simulator that models a 4-wide out-of-order processor. We extend ChampSim to simulate a Vector Unit with a vector register file that manages register dependencies and register renaming, and segmented vector processing units. We also consider memory address disambiguation and store-to-load forwarding in the load-store queue. The memory hierarchy models port contention, MSHRs, access latency, and bandwidth. Table 1 displays the parameters of the simulated architecture, which is based on contemporary vector architectures like the NEC SX-Aurora TSUBASA [22, 35].

Table 1: Architectural parameters of the simulated system.

Vector unit (VU)	
Issue / Commit width	4
Scheduler entries	64
Vector register length	16384 bits
VPU width (throughput)	2048 bits
Number of VPUs	4
Core frequency	2.4GHz
Memory subsystem	
LLC capacity	1 MB
LLC cache line size	64 bytes
LLC load-to-use latency	50 cycles
LLC associativity	8
LLC → VU bandwidth	2048 bits/cycle
LLC MSHR entries	256
Main memory bandwidth	256 GB/s
Main memory latency	60ns
Prefetchers	
Nextline	Aggressivity 8 lines
Best Offset Prefetcher (BOP)	Aggressivity 2 lines
Signature Path Prefetcher (SPP)	LLC confidence 20%
Percepton-based Prefetch Filtering (PPF)	Confidence threshold 15%
ReVeLA	Aggressivity 64, 16-entry STT
ReVeLA + Nextline	
ReVeLA + BOP	
ReVeLA + SPP	
ReVeLA + PPF	

4.2 HPC workloads

The selected benchmarks come from a wide range of domains: the PARSEC suite [8] (*blackscholes*, *canneal*, *streamcluster*, and *swaptions*), Rodinia [9] (*particlefilter* and *pathfinder*), Basic Linear Algebra Subroutines (BLAS) kernel (*trmm*) [37], the *somier* benchmark [27], 5 known scientific kernels (*axpy*, *cholesky*, *jacobi*, *MxM*, *spmv*), and five additional scientific workloads (*fft* [40], *lulesh* [20], *HPCG* [12], *resnet* [14], and the *RTM* isotropic kernel [45]).

Table 2 provides the arithmetic intensity of each considered workload and the input sets we consider. All codes have been vectorized using hand-tuned intrinsics based on the RISC-V "V" vector extension [36] and compiled with the LLVM [24] clang v17.0 compiler. Codes are Vector-Length Agnostic (VLA), which means they can run on architectures with different vector lengths without recompiling.

4.3 Evaluated Prefetchers

Our experimental campaign considers the following prefetching methods: the Next Line Prefetcher (*NextLine*) [30], the Best Offset Prefetcher (*BOP*) [26], the Signature Path Prefetcher (*SPP*) [21], the Perceptron-based Prefetch Filtering Prefetcher (*PPF*) [7] and

Table 2: Set of evaluated benchmarks, with their domain, inputs used and arithmetic intensity.

Benchmark	Domain	Computational pattern	Input used	Arithmetic intensity (a)
axpy	Scientific kernel	Dense Linear Algebra	524288 DP elems	0.099
cholesky	Scientific kernel	Dense Linear Algebra	1536×1536	3.689
spmv	Scientific kernel	Sparse Linear Algebra	bmw7st_1.mtx 141347×141347	0.142
blackscholes	Financial Analysis	Dense Linear Algebra	in_64k	5.611
canneal	Engineering	Unstructured Grids	nsw=100 t=300 netlist=2500000.nets ns=8	1.000
jacobi-2d	Engineering	Dense Linear Algebra	n=256 tsteps=2	0.142
particlefilter	Medical Imaging	Structured Grids	x=4096 y=4096 z=16 np=2048	144.708
pathfinder	Grid Traversal	Dynamic Programming	1024×1024	0.333
streamcluster	Data Mining	Dense Linear Algebra	k1=10 k2=20 d=128 n=chunk=16k cluster=1000	3.891
swaptions	Financial Analysis	MapReduce	ns=2 sm=32768	2.682
fft	Engineering	Fast Fourier Transform	256×256 blocksize=8	0.767
HPCG	HPC benchmark	Sparse Linear Algebra	nx=64 ny=64 nz=64	0.045
lulesh	Hydrodynamics	Unstructured Grids	i=2 s=30	0.897
resnet	Image Recognition	Convolutions	mb=1 ic=256 oc=1024 ih=14 oh=14 kh=1	1.565
RTM_iso	Engineering	Finite Difference Method	nit=3 nx=1040 ny=1040 nz=16	0.301
trmm	Scientific kernel	Dense Linear Algebra	1024×1024	3.692
MxM	Scientific kernel	Dense Linear Algebra	ix=iy=iz=256	5.564
somier	Physics simulation	Dense Linear Algebra	iter=4 N=20	0.229

a Calculated as a number of floating-point operations (FLOPs) divided by the number of bytes read (FLOPs/Byte).

the Register Vector Length Agnostic (*ReVeLA*) prefetcher that we describe in Section 3. We also consider combining ReVeLA with the other four considered prefetchers, *i.e.*, *ReVeLA + NextLine*, *ReVeLA+BOP*, *ReVeLA+SPP*, and *ReVeLA+PPF*. We consider the best configuration per each prefetcher for the HPC workloads we evaluate. Table 1 shows the parameters we use for each prefetcher. For example, the NextLine prefetcher considers an aggressivity of 8 cache lines, i.e., if a miss to an address @ takes place, the prefetcher brings to the cache the 8 consecutive 64-Byte cache blocks located right after the @ that triggered the miss. The four prefetcher combinations use the best parameterization per prefetcher when applied in isolation. We focus our evaluation on state-of-the-art prefetchers for lower level caches since vector units are typically interfaced to these lower levels, *i.e.*, memory accesses triggered by vector units typically bypass the L1D cache to avoid overwhelming the scalar data it may contain [22, 35]. Combining ReVeLA with the other four considered prefetches is possible due to the small ReVeLA area cost (436 Bytes), which is negligible compared to BOP (7.51KB), SPP (35.02KB), and PPF (39.34KB).

4.4 Details of the Evaluation Campaign

Our experimental campaign compares the performance achieved by all considered prefetchers with respect to a baseline system without any prefetcher. We describe all considered prefetchers in Section 4.3 and we show them in Table 1. Section 5.1 shows this performance evaluation. In our experiments we also show the coverage of each considered prefetcher and break it down in: (1) *useful* - the number of prefetched blocks accessed once the prefetch operation is completed, (2) *late* - prefetched blocks accessed while the prefetch request is still outstanding, and (3) *useless* - prefetch blocks evicted without being accessed. Section 5.2 displays our evaluation in terms of prefetch coverage. Our evaluation also displays the performance achieved by all considered prefetchers under different memory bandwidth and cache size scenarios in Section 5.3, and a sensitivity analysis of the ReVeLA prefetcher with respect to prefetching aggressivity and STT table size in Section 5.4.

5 EVALUATION

5.1 Performance Results

We evaluate the performance of each considered prefetcher by comparing its speed-up with respect to the baseline system that Section 4 describes without using any LLC prefetcher. Figure 9 shows the obtained results. The x-axis shows the considered workloads and the y-axis represents the speedup of each considered approach. In some scenarios, the speed-ups are larger than the maximum represented value. In these cases, we use boxes to represent the speed-up values. For example, Jacobi experiences speed-ups of 2.51×, 2.72×, 2.91×, 2.83×, and 2.71× when using ReVeLA, ReVeLA+NextLine, ReVeLA+BOP, ReVeLA+SPP, and ReVeLA+PPF, respectively. Our experiments show that ReVeLA, either applied alone or combined with the other considered prefetchers, provides significant performance speed-ups for many workloads, such as

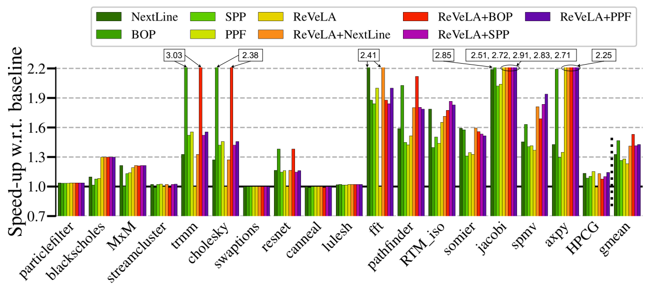


Figure 9: Speed-up of the different prefetcher configurations with respect to the baseline.

pathfinder and Jacobi, while it does not improve performance in others, such as swaptions, lulesh, and canneal. The reasons for ReVeLA not improving the performance of some workloads are twofold. Firstly, some workloads may fit entirely in the last-level cache (LLC) and therefore do not suffer from a high Misses Per Kilo-instruction (MPKI) rate, as is the case for the particlefilter workload. Secondly, some workloads do not display meaningful memory streams due to several reasons for this: (i) workloads may display random access patterns, e.g., canneal; (ii) they may use external libraries for which the compiler does not generate vector length agnostic code that ReVeLA can exploit, as is the case for resnet which uses libdnnl. Overall, ReVeLA achieves a geometric mean speed-up of 1.23× over all benchmarks.

Figure 9 shows that NextLine, BOP, SPP, and PPF achieve a geometric mean speed-up of 1.32×, 1.47×, 1.27×, and 1.28× respectively. NextLine's good performance can be attributed to the regular nature of some workloads such as axpy, while BOP's ability to select the best scoring offset per workload leads to a larger speed-up than NextLine. SPP provides a remarkable performance improvement compared to a system without any prefetching mechanism, but it is lower than that of NextLine and BOP due to the regular offsets displayed by linear algebra workloads (*e.g.*, jacobi, trmm), which match well with BOP. PPF achieves better performance than SPP for a couple of workloads (apxy and HPCG), although in general these two prefetches behaves similarly. Since PPF is an improvement of SPP that uses a neuron-based prefetch filtering approach, it provides better performance than SPP performance in some scenarios.

Despite the excellent performance improvements achieved by NextLine, BOP, SPP, and PPF, these prefetchers leave some performance on the table. Since ReVeLA uses a different logic to trigger prefetch requests than classical cache prefetchers and has a minimal area overhead, it can be combined with them. Figure 9 shows that when ReVeLA is combined with NextLine, the performance improves from 1.32× to 1.42× with respect to the baseline. In the case of BOP, the improvement goes from 1.47× to 1.53×, for SPP it goes from 1.27× to 1.41×, and for PPF it goes from 1.28× to 1.43×. Therefore, ReVeLA provides performance improvement of 6.57%, 4.46%, 11.83%, and 11.40% when combined with NextLine, BOP, SPP, and PPF, respectively, while requiring minimal area overhead. Exploiting the semantics of vectorized codes via a lightweight hardware extension enables these improvements.

5.2 Coverage Results

Figure 10 shows coverage data for all considered workloads and prefetching techniques that Section 4.3 describes. The y-axis shows percentage of LLC misses broken down into useful, late, or useless prefetches. Section 4.4 provides a definition of these categories. The addition of useful and late prefetchers indicates the percentage of cache misses that are served by prefetched cache lines. Figure 10 indicates that the average percentage of cache misses served by ReVeLA (45.64%) is significantly lower than NextLine (82.23%), BOP (66.63%), SPP (81.81%), and PPF (82.38%). However, ReVeLA's percentage of useful prefetches (40.70%) is higher than NextLine (30.05%) and PPF (34.09%), similar to SPP (40.20%), and smaller than BOP (53.93%). Additionally, ReVeLA displays a lower percentage of late prefetches (4.94%) than NextLine (52.17%), BOP (12.71%), SPP (41.60%), and PPF (48.28%). These data indicates that while ReVeLA does not find as many occasions to issue prefetch requests as NextLine, BOP, SPP, and PPF, it is timely when doing so. The main advantage of ReVeLA with respect to NextLine, BOP, SPP, and PPF is its negligible average percentage of useless prefetches (0.93%). In contrast, NextLine, BOP, SPP, and PPF suffer from 8.14%, 9.90%, 8.54%, and 7.38% average useless prefetches, respectively, which are one order of magnitude larger than ReVeLA. Most of these useless prefetches are triggered when running a set of particularly irregular workloads (i.e., canneal, HPCG and streamcluster), but there are also non-negligible numbers of useless prefetches on other workloads (i.e., particlefilter, RTM_iso, somier, spmv and swaptions).

The combination of ReVeLA with NextLine, BOP, SPP, and PPF increases coverage by converting late prefetches to useful ones for most workloads (*e.g.*, axpy, blackscholes, jacobi, MxM and particlefilter). For example, the average percentage of useful prefetches experiences an increase of 11.09% when using the combined ReVeLA+BOP compared to using BOP alone. In a few scenarios, combining ReVeLA with another prefetcher reduces the number of useful prefetches (i.e., lulesh with BOP or RTM_iso with NextLine and SPP), but this effect does not affect performance in the case of lulesh and even improves it in the case of RTM_iso. This improvement is due to the additional prefetches generated by ReVeLA.

By considering both performance speed-up and coverage data together, we observe that some workloads that do not benefit from

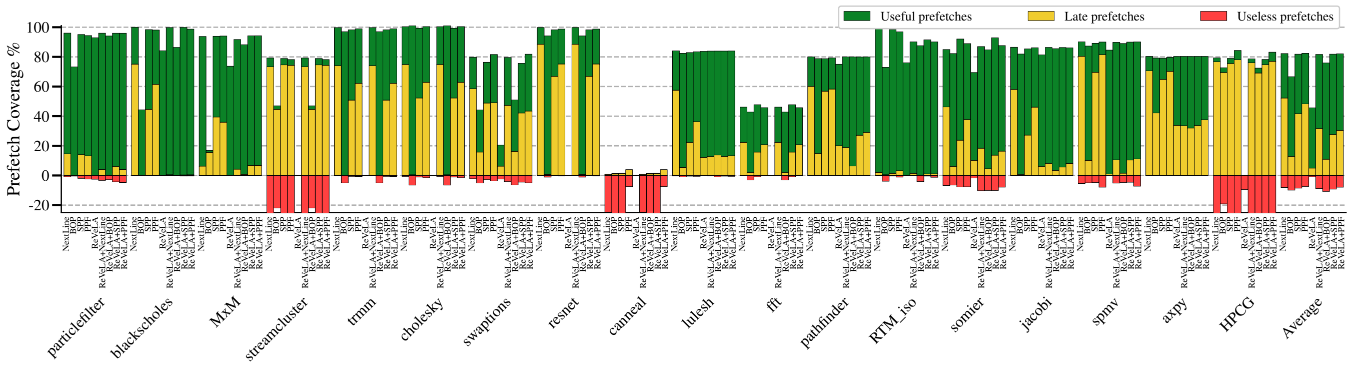


Figure 10: Prefetcher coverage of the different configurations.

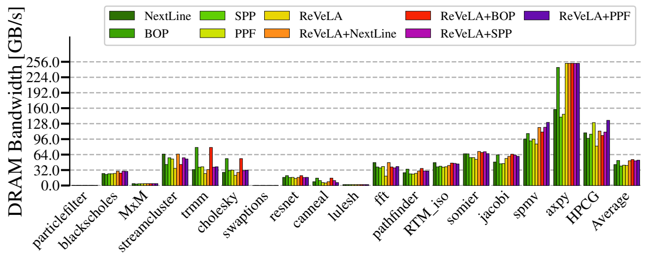


Figure 11: Effective DRAM bandwidth for different prefetcher configurations (including baseline) with DRAM bandwidth of 256 GB/s.

using a prefetcher with high coverage in terms of combined useful+late prefetches (e.g., lulesh, particlefilter, swaptions). This is due to the low LLC miss ratio experienced in the baseline scenario, less than 1% for lulesh, particlefilter and swaptions. In the case of streamcluster, its high percentage of useless prefetches (NextLine 78.16%, BOP 21.95%, SPP 57.71%, PPF 50.86%) undermines the benefits of the useful (NextLine 5.95%, BOP 2.43%, SPP 4.29%, PPF 3.99%), and late prefetches (NextLine 73.29%, BOP 44.57%, SPP 74.60%, PPF 74.20%).

5.3 Impact of DRAM Bandwidth and Cache Size

In this section, we analyze the impact of DRAM bandwidth and cache size on the performance of the considered prefetching mechanisms. Figure 11 presents the bandwidth consumption of all evaluated prefetchers, taking into account the system described in Table 2. The figure highlights that all workloads that benefit from prefetching also experience an increase in bandwidth consumption when prefetching mechanisms are applied. However, the combination of ReVeLA with NextLine, BOP, and SPP results in only a slight increase in memory bandwidth consumption compared to the scenario where these prefetchers are applied alone. For instance, NextLine alone consumes an average of 43.34 GB/s, whereas the combination of NextLine and ReVeLA leads to a minor 18.2% increase, totaling 51.23 GB/s. Similarly, BOP combined with ReVeLA spends only 3.74% more bandwidth on average than BOP alone. The increase for SPP and PPF is 26.32% and 24.60% respectively. These contained increases are due to the high accuracy of ReVeLA, which only fetches data that is eventually accessed by the running workloads, thereby avoiding unnecessary bandwidth consumption. This characteristic enables the combination of ReVeLA with any cache prefetching method without incurring significant costs in terms of area or bandwidth consumption.

To demonstrate the significant performance benefits of combining ReVeLA with state-of-the-art cache prefetchers, we evaluate their impact in a scenario with a small DRAM bandwidth of only 32 GB/s. Figures 12 and 13 illustrate the speed-up and bandwidth consumption with 32 GB/s DRAM bandwidth while keeping all other architecture parameters set as indicated in Table 2. Compared to the scenario with 256 GB/s DRAM bandwidth discussed in Section 5.1, the performance increase with prefetching mechanisms is lower. However, ReVeLA still manages to improve the performance when combined with NextLine, BOP, and SPP, even in this scenario

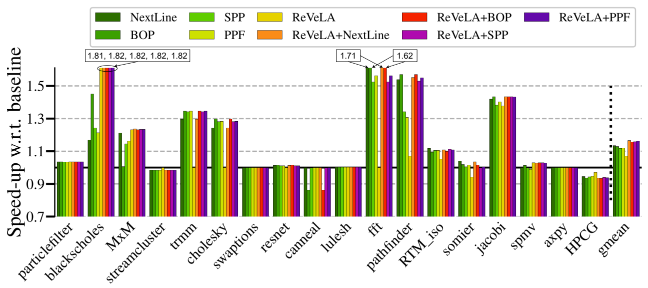


Figure 12: Speed-up of the different prefetcher configurations w.r.t. baseline with DRAM bandwidth of 32 GB/s.

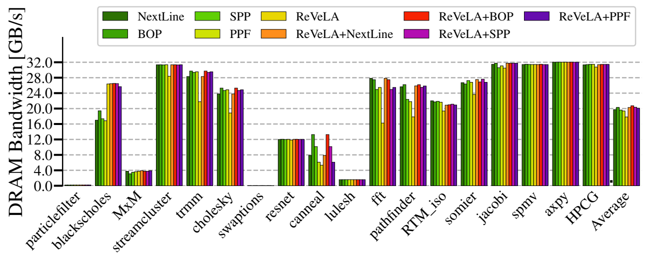


Figure 13: Effective DRAM bandwidth for different prefetcher configurations (including baseline) with DRAM bandwidth of 32 GB/s.

of limited DRAM bandwidth. In the case of NextLine, ReVeLA increases the speed-up from 1.13× to 1.17×, which corresponds to a 3.53% increase. Similarly, in the case of BOP, ReVeLA increases the speed-up from 1.13× to 1.16×, which also represents a 2.65% improvement. When combined with SPP, ReVeLA achieves a 3.68% improvement in performance. Finally, when combined with PPF, ReVeLA achieves a 3.85% increase in performance. These results demonstrate the effectiveness of ReVeLA in improving performance in scenarios with limited DRAM bandwidth.

Figure 14 illustrates the impact of combining ReVeLA with NextLine, BOP, and SPP in a high-bandwidth scenario (512 GB/s). These data reveal two major differences compared to the 256 GB/s scenario. Firstly, in the case of the axpy workload, ReVeLA achieves a larger improvement (2.73×) than in the 256 GB/s scenario (2.25×). Since axpy is a fundamentally memory-bound workload, ReVeLA benefits from a higher memory bandwidth. As in the 256 GB/s scenario, the performance of axpy is the same for ReVeLA, ReVeLA+NextLine, ReVeLA+BOP, ReVeLA+SPP, and ReVeLA+PPF; with improvements of 29.74% over BOP. Secondly, for the case of spmv, ReVeLA, ReVeLA+NextLine, ReVeLA+BOP, ReVeLA+SPP, and ReVeLA+PPF obtain speed-ups of 1.42×, 1.86×, 1.71×, 1.88×, and 2.03×, respectively, which are larger than the ones obtained for the 256 GB/s scenario: ReVeLA 1.37×, ReVeLA+NextLine 1.81×, ReVeLA+BOP 1.69×, ReVeLA+SPP 1.83×, and ReVeLA+PPF 1.94×.

Figure 15 displays performance data for all considered prefetching techniques and workloads in a scenario with a 16MB LLC, while all other configuration parameters are set as Table 2 describes. The benefits of using a prefetcher in any configuration are much lower than in the 1 MB scenario, as most workloads now fit in LLC. This

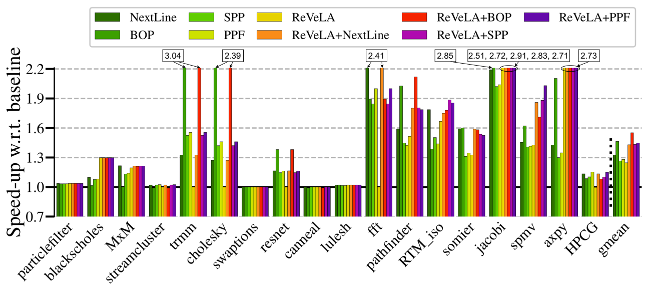


Figure 14: Speed-up of the different prefetcher configurations w.r.t. baseline with DRAM bandwidth of 512 GB/s.

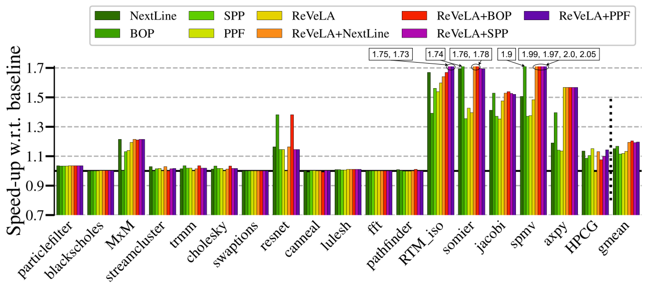


Figure 15: Speed-up of the different prefetcher configurations w.r.t. baseline with a 16 MB Cache.

is notable on blackscholes, blis, cholesky, fft and pathfinder. In some other benchmarks like axpy, jacobi and RTM_iso, the improvements of the different configurations are lower, but retain the same pattern of ReVeLA improving the other prefetchers. Remarkably, the speed-up on somier and spmv is even better that with the default 1 MB cache with all prefetchers. This is due to the fact that, while these workloads suffer a considerable number of useless prefetches with a 1 MB cache, this is no longer the case for a 16 MB cache. A large 16 MB cache avoids evicting prefetched cache lines that are eventually accessed by the workload, that is, useless prefetches become useful ones. In the 16 MB cache scenario, ReVeLA produces improvements of 3.70%, 3.14%, 6.95%, and 6.84% when combined with NextLine, BOP, SPP, and PPF, respectively.

5.4 Sensitivity Analysis of ReVeLA Configuration Parameters

In this section, a sensitivity analysis is performed to justify the design choices of ReVeLA, which include a 16-entry STT table, a 64 cache lines aggressivity limit, 8-bit LRU counters, and a 16-entry prefetch queue. Figure 16 displays the results of the analysis considering different aggressivity degrees ranging from 8 to 128 cache lines. All other architecture parameters are set as Table 1 indicates. Some workloads show a higher performance for higher aggressivity degrees (i.e., axpy and jacobi), while others reach their maximum performance with moderate aggressivity (i.e., MxM, pathfinder, and somier). Since all workloads except axpy already reach their maximum performance with a 64-cache line aggressivity, this value was chosen for ReVeLA.

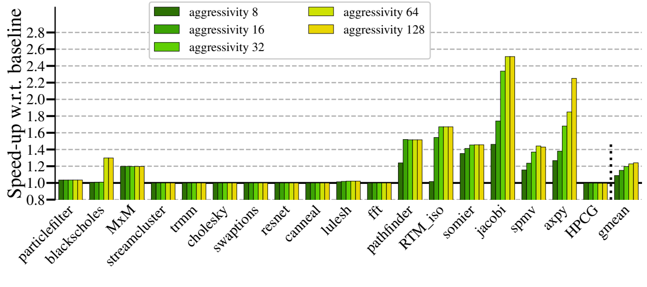


Figure 16: ReVeLA performance evaluation with different aggressivity parameterization (16 STT table entries).

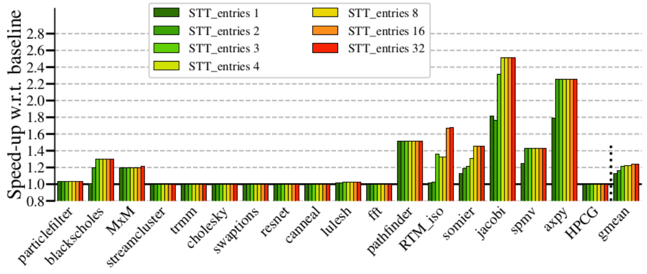


Figure 17: ReVeLA performance evaluation with different number of STT entries (Aggressivity 128).

Figure 17 shows the impact of increasing the number of STT entries. Our experiments consider 1-, 2, 3-, 4-, 8-, 16-, and 32-entry tables and set all other parameters as in Table 1. Data indicate that a 16-entry table provides the maximum performance for all workloads except MxM and RTM_iso, where a 32-entry table provides negligible improvements with respect to the 16-entry scenario. Therefore, we use a 16-entry entry table since it provides almost the same performance improvement as a 32-entry scenario while requiring half the area. Remarkably, ReVeLA produces close to maximum performance (91.3%) with a 4-entry STT table, which makes it possible to implement ReVeLA with a very small area overhead.

We also perform sensitivity studies regarding the number of bits devoted to LRU counters and the size of the prefetch queue. Regarding the LRU counters, we do not observe any performance benefit increasing their number of bits. Contrarily, we observe that ReVeLA delivers worse performance when using larger bit counts for the LRU counters since stall entries remain in STT for a longer period. Similarly, we do not observe any significant benefit in increasing the prefetch queue size.

6 RELATED WORK

Data cache prefetching is a technique to hide memory access latency by proactively fetching data blocks into the cache hierarchy in anticipation of demand requests from cores [3–7, 10, 16–18, 21, 26, 29, 30, 32, 42, 43]. These prefetching techniques can be divided into two categories: spatial and temporal prefetchers. Spatial prefetchers [4, 6, 16, 21, 26, 29, 32] exploit the similarity of access patterns between different memory regions, while temporal prefetchers [3, 17, 23, 42, 43] keep track of the sequence of past cache accesses to anticipate future misses, assuming a recurrence of those past accesses in the future.

Spatial prefetchers incur orders of magnitude less storage overhead than temporal prefetchers [4] since they keep track of deltas or offsets between accessed memory blocks, while temporal approaches store complete sequences of past cache accesses. For this reason, spatial prefetchers are widely used in industrial implementations [1, 11, 13, 31]. Spatial prefetchers also have the advantage of serving compulsory misses, which constitute a key bottleneck in some workloads [33], by exploiting observed deltas within already accessed pages to prefetch data corresponding to new pages. In contrast, temporal prefetchers are unable to predict future accesses when the application jumps to a different memory region. Additionally, previous work indicates that prefetch requests reaching DRAM triggered by spatial prefetchers typically trigger row buffer hits, in contrast to temporal prefetchers. These benefits in terms of row buffer hits reduce overall system energy consumption [4, 16, 41].

This paper proposes the use of ReVeLA to complement three state-of-the-art spatial prefetchers for lower level caches: the Best Offset Prefetcher (*BOP*) [26], the Signature Path Prefetcher (*SPP*) [21], and the Perceptron-based Prefetch Filtering Prefetcher (*PPF*) [7]. Combining ReVeLA with each one of these three prefetchers in-creases the number of useful prefetchers, while incurring small additional memory bandwidth and negligible area overhead. The combination of ReVeLA with spatial prefetchers such as BOP, SPP, or PPF improves prefetch timeliness and accuracy and leads to performance improvements in a large variety of HPC workloads, particularly the memory-bound ones.

BOP uses history-based predictors to determine the most effective prefetch distance, with a training period where possible strides are evaluated. BOP triggers prefetch requests once the training period achieves a target score. While BOP can be modified to trigger prefetch requests during the training period, this modification incurs significant area overhead and reduces BOP effectiveness on certain workloads. Since ReVeLA does not require a specific training period, it matches very well with the BOP design.

SPP selectively loads blocks of data into the lower-level caches. While SPP is a flexible approach able to deliver performance gains in a wide range of scenarios, it obtains the largest improvements when dealing with irregular patterns. Due to its need for evaluating a confidence for the prefetches and throttle itself, SPP may never achieve the aggressivity needed on workloads that access a low number of very long memory streams. Combining ReVeLA with SPP mitigates this drawback as this combination increases the SPP performance in scenarios where a low number of very long memory streams are accessed.

PPF increases the coverage of SPP without negatively impacting its accuracy. PPF enables more aggressive tuning of SPP, which increases coverage by filtering out the growing numbers of inaccurate prefetches such an aggressive tuning incurs. Still, PPF still suffers from similar issues as SPP for workloads accessing a low number of very long memory streams. Combining ReVeLA with PPF improves performance since the combination makes it possible to perform very aggressive and accurate prefetching for long memory streams.

Previous work proposes TLB prefetchers [19, 38, 39] to hide the latency of page walks and accelerate memory address translation. These previous approaches either prefetch page table entries located next to the one that triggered the TLB miss [38], are table-based prefetchers that correlate miss patterns with distances between virtual pages that produce consecutive TLB misses [19], or exploit page table locality to improve TLB prefetching [39]. Since ReVeLA works with virtual addresses, it may prefetch memory page translations into the TLB and, therefore, it can be combined with these previous approaches to improve their accuracy and timeliness in a similar way as it does with data cache prefetchers.

7 CONCLUSIONS

This paper proposes ReVeLA, the first approach able to leverage the program semantics of vectorized codes to guide data cache prefetching. ReVeLA is able to exploit the information contained in vector memory instructions to issue highly-accurate and timely prefetch requests. ReVeLA particularly shines when combined with a state-of-the-art prefetcher, since it significantly increases the number of useful prefetchers without incurring significant additional bandwidth consumption. In addition, the storage overhead of ReVeLA is very small (436 bytes when using a 16-entry SST table), which makes it possible to incorporate ReVeLA in general purpose vector processors with negligible cost.

Since vector processors require well-vectorized codes that are able to exploit long vector lengths, ReVeLA does not require any specific code modifications besides code vectorization. In addition, ReVeLA does not incur any performance slowdown for those codes that do not benefit from it, making it a low-cost hardware component that complements very well state-of-the-art prefetchers and boosts their performance when applied to a wide range of workloads. The low area cost, accuracy, timeliness, and ability to boost the performance of state-of-the-art data cache prefetchers, make ReVeLA a well-suited design to be incorporated to vector processors.

ACKNOWLEDGMENTS

This work has been partially supported by the Spanish Ministry of Science and Innovation MCIN/AEI/10.13039/501100011033 (contract PID2019-107255GB-C21) and ESF Investing in your future, the Generalitat of Catalunya (contract 2021-SGR-00763), the European HiPEAC Network of Excellence, and the European Processor Initiative (EPI), which is part of the European Union's Horizon 2020 research and innovation program under grant agreement No. 826647. A. Armejach is a Serra Hunter Fellow. The authors thank the Departament de Recerca i Universitats de la Generalitat de Catalunya for supporting the Research Group "Performance understanding, analysis, and simulation/emulation of novel architectures" (Code: 2021 SGR 00865).

REFERENCES

[1] [n. d.]. Intel Xeon Gold. https://en.wikichip.org/wiki/intel/xeon_gold/6258r.

[2] 2023. ChampSim. https://github.com/ChampSim/. Accessed: July 24, 2023.

[3] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino Temporal Data Prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 131–142.

[4] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo Spatial Data Prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 399–411.

[5] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1121–1137. https://doi.org/10.1145/3466752.3480114

[6] Rahul Bera, Anant V. Nori, Onur Mutlu, and Sreenivas Subramoney. 2019. DSPatch. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. https://doi.org/10.1145/3352460.3358325

[7] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. 2019. Perceptron-Based Prefetch Filtering. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 1–13.

[8] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[10] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.* 44, 5 (1995), 609–623. https://doi.org/10.1109/12.381947

[11] Pat Conway and Bill Hughes. 2007. The AMD Opteron Northbridge Architecture. *IEEE Micro* 27 (2007). https://doi.org/10.1109/MM.2007.43

[12] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2016. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *The International Journal of High Performance Computing Applications* 30, 1 (2016), 3–10.

[13] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya. 2020. Evolution of the Samsung Exynos CPU Microarchitecture. In *Proceedings of the 47th International Symposium on Computer Architecture*.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[15] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[16] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2009. Access Map Pattern Matching for Data Cache Prefetch. In *Proceedings of the 23rd International Conference on Supercomputing* (Yorktown Heights, NY, USA) *(ICS '09)*. Association for Computing Machinery, New York, NY, USA, 499–500. https://doi.org/10.1145/1542275.1542349

[17] Akanksha Jain and Calvin Lin. 2013. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 247–259.

[18] M. Kampe and F. Dahlgren. 2000. Exploration of the spatial locality on emerging applications and the consequences for cache performance. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*. 163–170.

[19] Gokul B. Kandiraju and Anand Sivasubramaniam. 2002. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proceedings of the 29th Annual International Symposium on Computer Architecture* (Anchorage, Alaska).

[20] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary Devito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles H. Still. 2013. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. In *2013 IEEE 27th International Parallel and Distributed Processing Symposium (IPDPS)*. 919–932. https://doi.org/10.1109/IPDPS.2013.115

[21] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A.L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783763

[22] Kazuhiko Komatsu, Shintaro Momose, Yoko Isobe, Osamu Watanabe, Akihiro Musa, Mitsuo Yokokawa, Toshikazu Aoyama, Masayuki Sato, and Hiroaki Kobayashi. 2018. Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 685–696. https://doi.org/10.1109/SC.2018.00057

[23] S. Kumar and C. Wilkerson. 1998. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th International Symposium on Computer Architecture*. https://doi.org/10.1109/ISCA.1998.694794

[24] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[25] Sally A. McKee. 2004. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers* (Ischia, Italy) *(CF '04)*. Association for Computing Machinery, New York, NY, USA, 162.

[26] Pierre Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 469–480.

[27] Cristóbal Ramírez, César Alejandro Hernández, Oscar Palomar, Osman Unsal, Marco Antonio Ramírez, and Adrián Cristal. 2020. A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures. *ACM Trans. Archit. Code Optim.* 17, 4, Article 38 (nov 2020), 30 pages. https://doi.org/10.1145/3422667

[28] Alejandro Rico, José A. Joao, Chris Adeniyi-Jones, and Eric Van Hensbergen. 2017. ARM HPC Ecosystem and the Reemergence of Vectors: Invited Paper. In *Proceedings of the Computing Frontiers Conference (CF'17)*. 329–334. https://doi.org/10.1145/3075564.3095086

[29] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H Pugsley, and Zeshan Chishti. 2015. Efficiently prefetching complex address patterns. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 141–152. https://doi.org/10.1145/2830772.2830793

[30] A.J. Smith. 1978. Sequential Program Prefetching in Memory Hierarchies. *Computer* 11, 12 (1978), 7–21. https://doi.org/10.1109/C-M.1978.218016

[31] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* 36 (2016). https://doi.org/10.1109/MM.2016.25

[32] S. Somogyi, T.F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. 2006. Spatial Memory Streaming. In *33rd International Symposium on Computer Architecture (ISCA'06)*. 252–263. https://doi.org/10.1109/ISCA.2006.38

[33] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. 2009. Spatio-Temporal Memory Streaming. *SIGARCH Comput. Archit. News* 37, 3 (jun 2009), 69–80. https://doi.org/10.1145/1555815.1555766

[34] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martínez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37 (May 2017), 26–39.

[35] Keichi Takahashi, Soya Fujimoto, Satoru Nagase, Yoko Isobe, Yoichi Shimomura, Ryusuke Egawa, and Hiroyuki Takizawa. 2023. Performance Evaluation of a Next-Generation SX-Aurora TSUBASA Vector Supercomputer. arXiv:2304.11921 [cs.DC]

[36] The RISC-V Foundation. 2020. The RISC-V "V" Vector Extension. https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf.

[37] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Softw.* 41, 3, Article 14 (jun 2015), 33 pages. https://doi.org/10.1145/2764454

[38] Steven P. Vanderwiel and David J. Lilja. 2000. Data Prefetch Mechanisms. *ACM Comput. Surv.* (June 2000), 26 pages. https://doi.org/10.1145/358923.358939

[39] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas. 2021. Exploiting Page Table Locality for Agile TLB Prefetching. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 85–98. https://doi.org/10.1109/ISCA52012.2021.00016

[40] Pablo Vizcaino, Filippo Mantovani, and Jesus Labarta. 2021. Accelerating FFT Using NEC SX-Aurora Vector Engine. In *Euro-Par 2021: Parallel Processing Workshops: Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers* (Lisbon, Portugal). Springer-Verlag, Berlin, Heidelberg, 179–190. https://doi.org/10.1007/978-3-031-06156-1_15

[41] Stavros Volos, Javier Picorel, Babak Falsafi, and Boris Grot. 2014. BuMP: Bulk Memory Access Prediction and Streaming. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 545–557.

[42] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Temporal Prefetching Without the Off-Chip Metadata. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 996–1008. https://doi.org/10.1145/3352460.3358300

[43] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Efficient Metadata Management for Irregular Data Prefetching. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*.

[44] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (mar 1995), 20-24.

[45] Hua-Wei Zhou, Hao Hu, Zhihui Zou, Yukai Wo, and Oong Youn. 2018. Reverse time migration: A prospect of seismic imaging methodology. *Earth-Science Reviews* 179 (2018), 207–227.