# Speculative Vectorisation with Selective Replay

Peng Sun
University of Cambridge
Cambridge, UK
ps702@cl.cam.ac.uk

Giacomo Gabrielli
Arm Research
Cambridge, UK
giacomo.gabrielli@arm.com

Timothy M. Jones
University of Cambridge
Cambridge, UK
timothy.jones@cl.cam.ac.uk

***Abstract*—While industry continues to develop SIMD vector ISAs by providing new instructions and wider data-paths, modern SIMD architectures still rely on the programmer or compiler to transform code to vector form only when it is safe. Limitations in the power of a compiler's memory alias analysis and the presence of infrequent memory data dependences mean that whole regions of code cannot be safely vectorised without risking changing the semantics of the application, restricting the available performance.**

**We present a new SIMD architecture to address this issue, which relies on speculation to identify and catch memory-dependence violations that occur during vector execution. Once identified, only those SIMD lanes that have used erroneous data are replayed; other lanes, both older and younger, keep the results of their latest execution. We use the compiler to mark loops with possible cross-iteration dependences and safely vectorise them by executing on our architecture, termed selective-replay vectorisation (SRV). Evaluating on a range of general-purpose and HPC benchmarks gives an average loop speedup of 2.9×, and up to 5.3× in the best case, over already-vectorised code. This leads to a whole-program speedup of up to 1.19× (average 1.06×) over already-vectorised applications.**

## I. Introduction

Industry support for SIMD vector ISAs continues apace, with both Intel and Arm developing and extending their instruction sets and vector widths. The AVX-512 ISA from Intel [12] extends operations to work on 512 bits of data at a time while Arm has developed its Scalable Vector Extension (SVE) architecture [23] that allows code to be compiled once and run on a variety of implementations that each choose their own vector width, up to a maximum of 2,048 bits. Modern compilers provide support for automatic vectorisation of applications, with the trend to provide more sophisticated analysis and transformations to increase the amount of code that is vectorised and better assess the profitability of vectorising [25].

However, despite the presence and history of vectorisation techniques in both hardware and the compiler, in practice, aside from in certain domains, they are rarely used to vectorise general-purpose applications. This is partially due to the inability of compilers to perform accurate interprocedural pointer disambiguation and array dependence analysis, which limits the application of existing compiler and architectural vectorisation techniques. Current SIMD ISAs only provide support for operating on vectors of data when the transformation has been proven to be safe. In other words, the compiler or programmer must ensure that they do not violate any data dependences when converting the code from scalar to vector form, otherwise they risk changing the semantics of the application. This can be a time-consuming and tedious process for developers; for compilers, current static analyses are not able to accurately disambiguate pointers due to their need to be conservative [4, 10, 18, 19, 27], even though existing research shows that a pair of memory accesses rarely alias until and unless aliasing is obvious [8]. Further, the requirement for all code to be free of memory-dependence violations when vectorised means that irregular or infrequent dependences prevent whole regions of code from using the SIMD hardware, limiting the performance available.

Researchers have addressed this problem in two ways. Baghsorkhi et al. [2] proposed FlexVec, which adds run-time checks on memory dependences during vectorised code execution. When a violation occurs, they predicate off the erroneous lanes, providing partial vectorisation for only the first few "safe" lanes. Kumar et al. [13] use a binary translator to speculatively vectorise sequential code at run time, restarting at a checkpoint and falling back to the sequential version when violations occur. Both solutions incur run-time overhead, and leave performance on the table when the majority of the code after the dependence violation is parallel.

This paper takes a hardware approach. We speculatively execute regions of vectorised code and monitor the memory addresses accessed to identify memory-dependence violations. At the end of the region we re-execute the code for only those lanes that have obtained the wrong data. Older and younger lanes with correct data need not rerun. Using our selective-replay vectorisation architecture (SRV), which augments a standard out-of-order pipeline, we limit the overheads of detecting and correcting dependence violations, allowing compilers to vectorise freely, even in the presence of unknown dependences. Evaluation on a system with 16-element-wide vector operations (agnostic of element size) shows an average speedup of 2.9×, and up to 5.3× in the best case, across a range of SPEC CPU 2006 and HPC applications.

## II. Motivation

Modern compilers fail to vectorise code in the presence of unknown or infrequent data dependences. For example, listing 1 shows a loop with an indirect memory access where the values of array `x` are not known at compile time. Therefore, since the compiler cannot prove that `a[i]` and `a[x[i]]` always refer to different memory locations, this loop is not

```
1 /* Read integers from the standard input. */
2 int *x = read();
3 for (i = 0; i < N; i++) {
4   a[x[i]] = a[i] + 2;
5 }
```

Listing 1: Example code with read-after-write cross-iteration dependences every four iterations when `read()` returns $\{3, 0, 1, 2, 7, 4, 5, 6, 11, 8, 9, 10...\}$.

vectorised. Assuming that the `read()` function provides values $\{3, 0, 1, 2, 7, 4, 5, 6, 11, 8, 9, 10...\}$, then a read-after-write memory-dependence violation does indeed occur if vectorising with four or more lanes, because `a[3]` reads a value before `a[x[0]]` has updated it, obtaining the wrong data. This pattern repeats every four iterations. In this case, the compiler is correct in reasoning that it is unsafe to vectorise this loop unless it knows that fewer than four lanes will be used or there is a method of detecting and correcting the violations that occur.

We performed limit studies across the benchmarks used in this paper (more details of the applications in section V) to examine how vectorisation of general-purpose and HPC workloads are affected by unknown dependences. We instrumented each workload to record through-memory dependences at run-time and estimated the optimal performance that vectorisation could obtain for inner loops if it failed to vectorise only in the presence of true dependencies. To achieve this, we emulated vectorisation in groups of 16 iterations at a time for these loops, assuming that logic would be available to buffer stores so as to avoid dependence violations from false (WAW and WAR) dependences, and additional logic to detect early loop exit (e.g., through a `break`). Results showed an average of 2.1× potential whole-program speedup if we could vectorise all inner loops, but this dropped to an average of only 1.02× speedup if we could not vectorise those with unknown through-memory dependences. More than 70% of the currently unvectorised inner loops have these types of dependences, therefore techniques that can address this issue are critical.

One existing option to achieve this is by using FlexVec [2]. This adds compiler-generated run-time checking instructions at the start of the loop and vectorises as many lanes as possible, up to the lane with the dependence violation. In the case of listing 1, assuming 16 vector lanes, FlexVec first forms a loop to check the memory dependences between `a[i]` and `a[x[i]]` within each iteration and marks lanes 3, 7, 11, and 15 to indicate dependence violations. It then partially vectorises the first 16 iterations into five groups—executing lanes 0–2 in the first iteration, 3–6 in the second, 7–10 in the third, 11–14 in the fourth, and 15 in the last.

However, there are two drawbacks to this approach. First, run-time check instructions, usually arranged in a separate loop, introduce high run-time overhead. For the example code in listing 1, FlexVec implements a `VPCONFLICTM` instruction to perform explicit memory disambiguation. This instruction compares each element of `a[i]` with all enabled previous elements of `x[i]` until violation is detected. The overhead of these comparisons increases when the number of memory accesses that may alias with each other increases.

Second, partial vectorisation does not fully exploit the potential of data-level parallelism. For the above example, ideally we can vectorise and execute this loop in just two iterations. During execution of the first iteration, we monitor the memory addresses accessed and dynamically mark the lanes that cause a dependence violation (3, 7, 11, and 15). The second iteration re-executes just these lanes and, as there are no more dependence violations, finishes execution.

We propose SRV to address both of these drawbacks and resolve unknown or periodic dependence violations from vectorisation using this mechanism. The following sections describe how selective-replay vectorisation works, as well as the hardware and compiler support required, before we evaluate its performance in section VI.

## III. Selective-replay vectorisation

SRV is a hardware vectorisation technique that allows the compiler to speculatively vectorise code, catching data-dependence violations automatically and re-executing affected lanes so as to maintain the sequential semantics of the program. This means the compiler can vectorise code even when it cannot prove the absence of data dependences between the SIMD lanes. New hardware monitors the memory addresses accessed by each lane to identify those that have read the wrong data; these are executed again to obtain their correct values. This allows the compiler to vectorise code with periodic or rare dependences so that it can also obtain the benefits of vectorised execution, safe in the knowledge that the hardware will catch all violations and resolve them.

Figure 1 shows an overview of a standard out-of-order superscalar pipeline with shaded blocks where we add logic for SRV. We assume a baseline architecture similar to Arm's SVE, with predicate registers to mask execution of lanes where necessary. SIMD instructions that execute under SRV proceed through the pipeline as normal, but have execution on each lane guarded by an additional predicate register, called the *SRV-replay register*, which is initially fully set (so all lanes execute each instruction). Memory-access instructions are buffered in the load-store queue to enable memory disambiguation, identifying cross-lane data-dependence violations. Violations are resolved by store-to-load forwarding, selective memory updates, or recording incorrect lanes in another predicate register, the *SRV-needs-replay register*, depending on the type of data-dependence violation. At the end of a sequence of vector instructions, the SRV-needs-replay register indicates whether any lanes need to be replayed. If so, it is copied to the SRV-replay register and execution jumps back to the start of that vector-instruction sequence.

### A. Enabling SRV execution

To support vectorisation using SRV, we introduce two new instructions: *SRV-start* and *SRV-end*. These are placed at the beginning and end of a vectorised loop body with unknown,

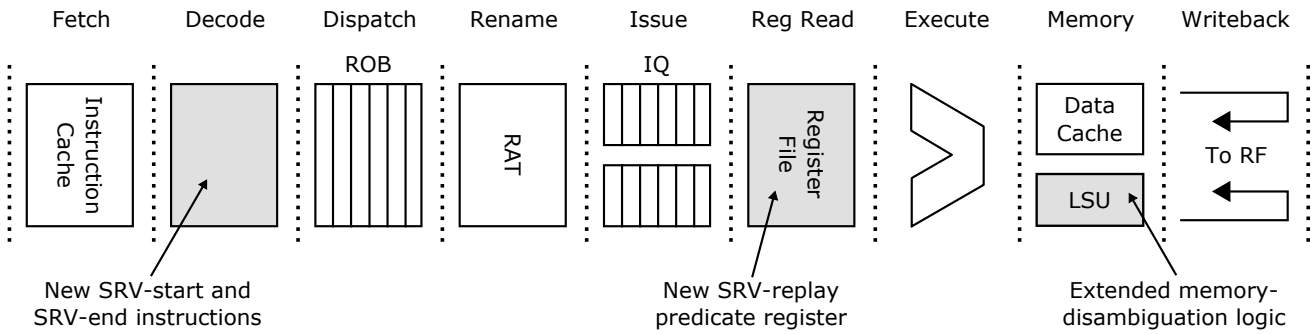


Fig. 1: Logic for SRV added to a standard out-of-order superscalar pipeline.

periodic or rare dependences, as shown in listing 2, pseudo-code that represents the final compiler-generated code from listing 1. We term the region bounded by SRV-start and SRV-end instructions the *SRV-region*, and it contains only vector instructions. SRV could also be used to vectorise non-loop code with unknown dependences, through the SLP algorithm [14], for example, although we do not pursue that direction in this paper. Inside the SRV-region, SRV is enabled; additional logic that is only required for SRV can be power gated when code outside an SRV-region is executed.

```
1 Loop:
2   srv_start
3   v_load  v0, a[i:i+15]
4   v_add   v0, 2
5   scatter v0, a[x[i]:x[i+15]]
6   srv_end
7   inc     i, 16
8   comp    i, N
9   bne     Loop
```

Listing 2: Pseudo-code for listing 1 using SRV.

The SRV-start instruction denotes the start of a speculative region and the point at which execution should restart if any lanes violate memory dependences, when they need to roll back and re-execute. Executing SRV-start involves recording the PC of the next instruction, fully setting the SRV-replay register and directing the load-store queue to perform extended memory disambiguation, described in section III-B. Scalar operations that may change the architectural state of a loop's iteration (e.g., induction variables) are kept outside the SRV-region by the compiler so that these variables remain non-speculative. Stored data from speculative lanes cannot leave the core and enter the cache hierarchy until they become non-speculative. Note that no checkpoints are taken of architectural registers (apart from the PC)—any changes made to registers during the SRV-region persist across re-executions.

Execution in SRV proceeds through the SRV-region in the same way as through regular code. On completion, when encountering an SRV-end instruction, if there have been no memory-dependence violations (indicated by a fully unset SRV-needs-replay register), then all speculative state can be committed and execution continues beyond the SRV-region. However, if there are any incorrect lanes, then execution rolls back to the point immediately after the previous SRV-start instruction, using the recorded PC, and only those lanes that have used incorrect data are replayed. Rollback may occur multiple times before all lanes are correct, but is bounded by the number of vector lanes (so for $N$ lanes, roll back can only happen at most $N-1$ times, but in section VI we show it causes execution of fewer than 1% additional iterations in the worst case for our workloads). SRV-regions cannot be nested, meaning that a second SRV-start cannot occur before an SRV-end instruction has been executed.

The SRV-start instruction carries an attribute that indicates the iteration ordering property of the SRV-region. If the lane number of a vector register increases when the accessing memory address increases (e.g., a loop with an increasing induction variable), the SRV-start instruction carries an *UP* attribute. Memory disambiguation works as described in section III-B. Otherwise, if the lane number of a vector register increases when the accessing memory address decreases (e.g., a loop with a decreasing induction variable), the SRV-start instruction carries a *DOWN* attribute. In this case, memory-address comparisons during memory disambiguation are adjusted to account for the changed direction of data access. This attribute is passed to the SRV-start instruction from the compiler.

### B. Memory disambiguation

To perform memory disambiguation in SRV, we leverage existing load-store unit (LSU) components within out-of-order superscalar cores, augmented with modest additional logic, which is only powered on when required, to capture data-dependence violations. We focus here on an LSU targeting a relaxed memory-consistency model aligned to the Armv8-A with SVE memory model [1]. We assume that the LSU is split into a load queue (LQ), store-address queue (SAQ) and store-data queue (SDQ). Contiguous and broadcast vector load instructions occupy only one entry in the LQ, whereas a vector gather takes up one entry for each lane that is loaded. Likewise, contiguous vector stores take up a single entry in

the SAQ and SDQ, whereas a vector scatter takes up one entry for each lane that has data stored.

*1) Load execution:* When a load instruction issues, it is compared with all older stores in the SAQ. When executing scalar code, if the load accesses an address that was written by an older store in the SAQ, data is forwardable from the corresponding SDQ to this load instruction. Although most architectures only allow store-to-load forwarding if an older store contains all data for the younger load, some implementations are more aggressive and enable partial store-to-load forwarding where a younger load obtains forwardable bytes from multiple entries of the SDQ and the cache [26]; we choose to do this.

However, with SRV, if the address accessed by the issuing load overlaps with that of an older store in a later lane, then a write-after-read (WAR) violation occurs. In this case, bytes with matched addresses are not forwardable from the matching SDQ entries because they represent data that would have been stored later in program order, in non-vectorised code. The load instruction instead must obtain forwardable data only from earlier lanes (or the same lane) of older SDQ entries. If no matches are found, it must load from the memory hierarchy, using a similar mechanism to that described by Witt [26]. Depending on the addresses matched, data can be read from both SDQ entries and corresponding cache lines concurrently, then combined to write into the destination vector register. This disambiguation scheme allows data to be partially loaded from the memory system and partially forwarded from the SDQ.

*2) Store execution:* When a store instruction issues, or when its virtual address becomes known, it is compared against all LQ entries that contain younger loads. In parallel, a search of the SAQ is required to detect write-after-write (WAW) violations.

For scalar code, a read-after-write (RAW) dependence violation occurs when a younger load has previously read data from an address that at least partially overlaps with the address written to by an issuing store. Whenever this happens, a signal is generated to squash this younger load and all instructions after it. In addition though, in SRV, RAW violations also occur when an issuing store writes to an address that at least partially overlaps with the address read by an older load in a later SIMD lane. In this case the older load has already read the wrong data and so it also needs to be squashed. However, there is no need to squash the whole instruction. Instead, SRV resolves the violation by recording the lane(s) in each prior load that violated this dependence. At the end of the SRV-region, only these lanes are replayed so that the later lanes with older loads pick up the correct data.

WAW violations occur when an issuing store writes data to the same address as written by an older store in a later lane. SRV resolves this issue by once more recording the violating lanes, then using this information to selectively update memory once the SRV-region is complete. In other words, it records all data stored during the SRV region but only writes the latest version of each address back to memory.

*3) Resolving dependences:* As described in previous paragraphs, data dependences may be violated by execution under SRV. WAR and WAW violations are handled immediately: in the former case, data is not forwarded from older stores in later lanes; in the latter case, the SAQ records which lanes of each store should write data to memory so that the most recent version (in program order) is eventually written back. RAW dependences, in contrast, are handled by setting bits in the SRV-needs-replay register and re-executing violating lanes when reaching the end of the SRV-region.

### C. Control flow

The SRV architecture supports arbitrary forwards control flow within an SRV-region through if-conversion. This means that code with if-then-else statements can be executed under SRV by creating predicates from the branch condition checks, removing branches, and executing the code under control of the predicates. However, it does not currently support loops or function calls due to the difficulties in calculating the number of LSU entries required when loops do not have a fixed number of iterations and call graphs contain recursion.

Each memory-access instruction within the SRV-region obtains an identifier, called an *SRV-id*. Memory instructions with the same PC are assigned the same SRV-id, (e.g., when splitting a scatter). During replay, no further entries are allocated in the LSU; instead, entries with the same SRV-id are updated.

To ensure this in the presence of predication, on the first execution of an SRV-region all instructions are fetched and issued. Lanes where the predicate is on are executed; those where it is off are unchanged. This creates entries in the LSU for all possible memory accesses, meaning that on replay, should one or more lanes execute instructions with a different predicate to before, they will not have to add entries to the LSU. This does not affect the number of times code is executed within the SRV-region, but means that instructions where the predicate is completely unset must be issued and allocate LSU entries the first time through the SRV-region.

This process only needs to be performed once when a new SRV-region is encountered. The microarchitecture records the LSU entries needed for the last SRV-region seen and, when it sees that region again, can pre-allocate the entries. The majority of SRV-regions are contained within a loop body, so they are likely to be executed many times consecutively.

### D. Other architectural changes

This section describes other changes to the architecture as a result of SRV.

*1) Serialisation point:* The SRV-end instruction creates a serialisation point, meaning that it will only be executed once all previous instructions have completed execution and, further, no younger instructions are allowed to execute before them. This is due to the SRV-needs-replay register containing "sticky" bits; a common method to handle these is to rely on a completion table, which necessitates serialisation. In order to achieve this, SRV-end is marked as non-speculative and the issue logic stalls the issue of younger instructions until this SRV-end is executed.

*2) Architectural state:* The architectural state of the processor is augmented with the SRV-replay register and the PC of the instruction following the SRV-start. Outside an SRV-region this PC value is set to 0x0 to indicate normal execution; inside, the oldest lane with its bit set in the SRV-replay register is non-speculative and will not be re-executed. Thus architectural state advances with each instruction committed from the SRV-region and on execution of an SRV-end instruction, when the SRV-replay register may change or the SRV-region finishes.

When a context switch or processor mode change occurs within the SRV-region (i.e., through an interrupt or exception), the current PC, SRV-replay register, and PC of the instruction following the SRV-start are sufficient to capture the exact point execution should return to. All non-speculative data in the LSU is written back to memory at this point (i,e., data corresponding to the oldest lane set in the SRV-replay register—up to the current PC—and all data from older lanes); speculative content is discarded. On resumption, the two PCs are restored. However, only the bit corresponding to the oldest lane in the saved SRV-replay register is restored, with bits in the SRV-needs-replay register set corresponding to all younger lanes. This avoids correctness issues that could occur if resuming execution of speculative lanes in the middle of the SRV-region by only resuming execution of the non-speculative lane until the SRV-end instruction is encountered, at which point all younger lanes will execute the entirety of the SRV-region.

*3) Interrupts and exceptions:* Interrupts and context switches that occur during an SRV-region are handled immediately as described in section III-D2. Exceptions are handled similarly to interrupts, except we first identify the lane number that caused the exception and handle it only if that lane is the oldest lane currently executing. If not, then this and all subsequent lanes are marked for re-execution, to guard against exceptions occurring as a result of using erroneous data after a memory-dependence violation. Through these methods, precise interrupts and exceptions are maintained.

*4) SAQ modification:* Each entry of the store queue is augmented with a speculative flag, which is set by store instructions within the SRV-region. Stores can commit when they reach the head of the ROB, but their data remains in the store queue while the speculative flag is set. Execution of the SRV-end instruction, if the SRV-needs-replay register is unset, clears all speculative flags and writes the stored data back to the L1 cache (subject to the constraints described in section III-B3).

*5) Register renaming:* Upon re-execution, instructions have a "merging" behaviour, in order to preserve the data of inactive (already executed) lanes. With renaming, this is typically achieved by reading the old destination register as an extra source operand to each instruction. When instructions write into new physical registers, they also need to read the old destination physical registers as source operands to merge the new and old data together.

*6) In-order architectures:* Applying SRV to an in-order processor is more straightforward than for an out-of-order machine, since the out-of-order execution of loads and stores by the baseline microarchitecture does not occur. In many ways, however, adding SRV is akin to adding a limited form of out-of-order execution to an in-order CPU, and still needs logic to detect data-dependence violations. To achieve this, we simply add an LSU to a standard in-order processor pipeline, with the SRV extensions described in section III-B.

*7) LSU overflow:* Memory instructions in the SRV-region should be kept in LSU entries until they are no longer speculative. However, the compiler may create an SRV-region with more loads and stores than available entries. To resolve this, we take the same approach as industrial transactional-memory schemes [28] that are supported on a best-effort basis, whereby transactions may fail at any point. In the context of SRV, should there be too many memory-access instructions, then we transparently fall back to sequential execution. Here, the SRV-region is repeated once for each lane, with only the oldest executed and committed each time.

Although not knowing the number of LSU entries will not affect the correctness of execution, blindly applying the code transformation could lead, in some corner cases, to slowdowns due to the vector version of the code not being as efficient as sequential code, especially when dealing with very large loop bodies. To address this, we propose two different solutions. First, the compiler could generate a sequential version of the loop code, to be executed in cases where the microarchitecture does not have enough LSU entries to support the vectorised version. Similar run-time checks are already inserted by compilers to circumvent other auto-vectorisation obstacles, e.g., for determining run-time aliasing of pointers. Second, the number of LSU entries in the target microarchitecture could be exposed to the compiler. The compiler would analyse the maximum number of memory accesses within each loop that it intended to vectorise with SRV, and only perform vectorisation if it could guarantee that the number of memory accesses would not cause overflow. Compilers already perform optimisations and code generation to target specific microarchitectures, with this typically specified through a "`-mcpu`" or a "`-mtune`" command-line option.

*8) Vector register-file ports:* In principle, SRV would require an additional vector register-file port for each vector-operation issue slot in order to support merging predication (i.e., predication that leaves inactive lanes untouched). In fact, supporting merging predication efficiently in an out-of-order microarchitecture is usually implemented by propagating the old value of the destination register onto a new physical register, implying an additional vector-register read. However, we can reasonably assume that a balanced baseline microarchitecture supporting latest-generation SIMD extensions, such as Arm SVE and Intel AVX512, which already supports merging predication, would already provide an adequate number of ports for handling merging operations at full throughput. Based on that assumption, the main additional overhead introduced by SRV is reduced to combining the general predicate associated with each vector-merging instruction with the SRV-replay register using a simple binary logic operation.

```
1 srv_start
2 v_storeb  v0, x[i:i+15]    // Instruction A
3 ...
4 v_loadb   v1, x[i:i+15]    // Instruction B
5 ...
6 v_loadb   v2, x[i+8:i+23]  // Instruction C
7 srv_end
```

Listing 3: Example pseudo-code to illustrate vertical and horizontal dependences between instructions. Instruction B has a vertical dependence with instruction A; C has a horizontal and vertical dependence with A.

### E. Use of transactional memory

Transactional memory detects memory violations between threads at run time and resolves them through re-execution. Applying it to vector execution, each SIMD lane could be viewed as a thread, with a strict ordering between them, similar to how the IBM Blue Gene/Q operates [20], and transactional memory used to detect and resolve the conflicts between lanes. However, unless the transactional memory system kept versions of each cache line, then it would have to re-execute lanes on WAR dependence violations, as well as RAW, to ensure correct execution in all situations.

### F. Summary

We have presented SRV, a technique for speculative vectorisation, describing how vectorised code executes on an out-of-order core, how memory-dependence violations are identified, and techniques to handle corner cases. The next section takes this forward to outline a detailed SRV microarchitecture.

## IV. SRV MEMORY-DISAMBIGUATION MICROARCHITECTURE

The key component of SRV is the ability to identify memory-dependence violations and correct them. To achieve this, we add additional logic into the load-store unit for memory disambiguation. We start with an example to present our terminology, before describing what happens on execution of load and store instructions.

### A. Terminology

We extend the concept of conventional, inter-statement data dependences, first introduced by Banerjee [4], with notations of lane numbers in vector instructions. In order to distinguish our definition of the inter-lane dependences from that of the inter-statement dependences introduced by Banerjee [4], we name the former horizontal dependencies and the latter vertical dependencies.

Listing 3 gives an example. Here, we show an SRV-region with three key instructions—two vector loads and a vector store, all operating on byte-length data in each lane. The vector store at line 2, instruction A, writes data to sixteen contiguous elements of the `x` array, indexed by the current iteration number. The vector load at line 4, instruction B, reads data back from the same elements. This corresponds to a *vertical* dependence between the two instructions, as would be the case in standard execution. However, the vector load at line 6, instruction C, loads data from sixteen contiguous elements of the `x` array, offset by eight elements from the current iteration number. Therefore, there is a dependence between different lanes of this load and the store at line 2. Here, lane 0 in instruction C accesses the same address as lane 8 in instruction A, lane 1 in C the same as lane 9 in A, and so on. This corresponds to a *horizontal* dependence between the two instructions, which is new to our technique.

Conventional dependence representations do not consider any lane information since only independent lanes will be formed into vector statements. However, in SRV operation, when vector statements may be formed without considering dependences, lane information is essential for dependence detection.

In addition to the new horizontal dependences, we define the address-alignment region as the address space that aligns with the vector length of a specific vector architecture. For example, if the vector architecture under consideration has a vector length of 64 bytes, the address spaces 0x00–0x3F, 0x40–0x7F, etc, are known as alignment regions. The address space 0x0C–0x4C spans two consecutive alignment regions. The start address of the alignment region is known as the address-alignment base.

### B. Vertical disambiguation mechanism

To identify vertical data dependences in the baseline microarchitecture, as well as in SRV, the LSU contains the logic shown in figure 2 marked by dashed lines and titled “Vertical Disambiguation Logic”. This consists of a bit vector per row in the SAQ and LQ, computed during issue, corresponding to the bytes accessed by the corresponding SAQ / LQ entry relative to the address-alignment base (the *bytes-accessed bit vector*), and a further bit vector that determines the bytes overlapped with each issuing load or store (the *VOB bit vector* or *vertically overlapped bytes bit vector*).

Figure 3 shows how this works for our example code in listing 3. The vector store is in the SAQ and SDQ and the first vector load (instruction B from line 4) issues. We assume here that the address-alignment base for the current dynamic instance of each instruction is 0xAB00 and the data is found at an offset of 16 bytes. As mentioned previously, each instruction operates on single bytes in each lane, so the Elem field is set to 1 and Size set to 16. Type and Lane fields are not important for this example.

When instruction B issues, the SAQ is consulted and a match occurs on the entry containing instruction A, since they have the same address-alignment base. The bytes-accessed bit vector for both instructions is the same, so the VOB bit vector consists of all bits set for the overlapping memory addresses. In this case, that is bits 16 to 31—this is because there is an offset of 16 bytes from the address-alignment base and then 16 bytes that are common between the two instructions. This indicates that all data for the load has been written by the prior store, so is forwardable from it.

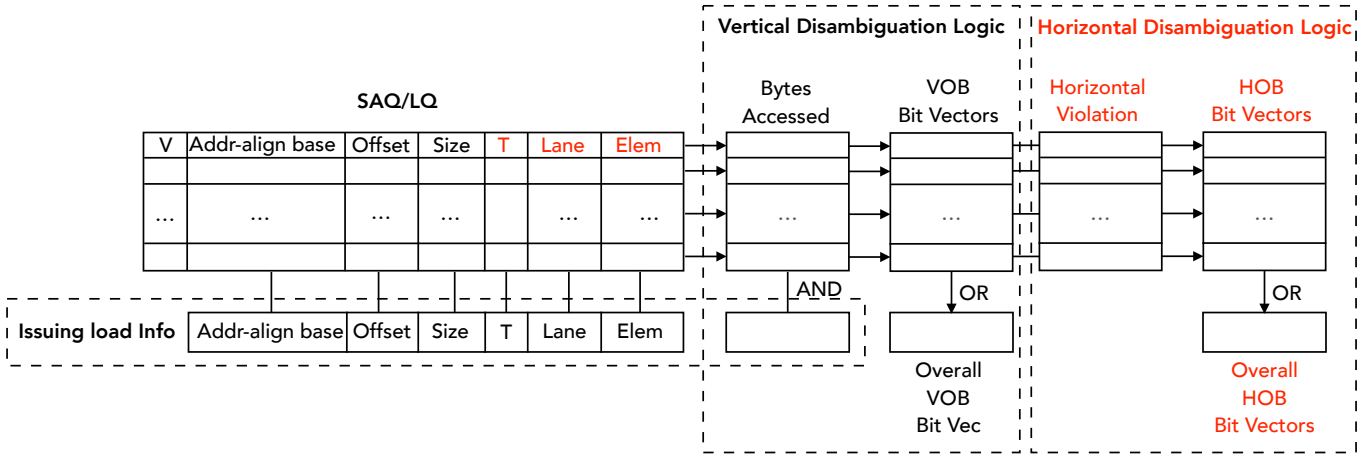


Fig. 2: Detailed vertical and horizontal disambiguation logic structure. When an issuing instruction searches the SAQ/LQ, bit vectors are generated to either determine the data that can be forwarded from the SDQ or generate control signals.

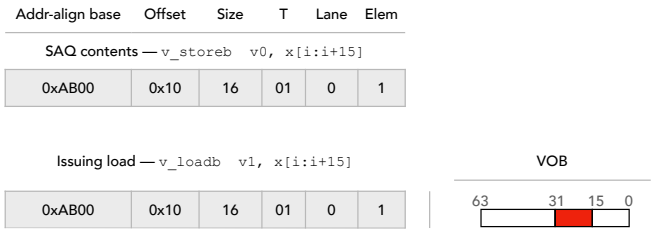


Fig. 3: Vertical disambiguation for the code in listing 3 when the vector load at line 4 issues. The red block in the VOB indicates bits set to 1.

In more detail, when a load instruction issues, its bytes-accessed bit vector is calculated and ANDed with the bytes-accessed bit vector of each matching row in the SAQ (i.e., with the same address-alignment base), with results written into the VOB bit vectors. All VOB bit vectors are ORed together to form the overall VOB bit vector, which indicates the bytes of the issuing load instruction that have been written by prior stores, whereas unset bits indicate bytes to read from memory. This action is managed so that younger stores are given priority to forward data, as described by Witt [26], if multiple SDQ entries contain data for the load instruction. Data forwarding from multiple SDQ entries will only occur when a contiguous load needs to get data from previous stores under write-after-read violation. We modelled this data grouping being completed within a single cycle, as explained by Witt [26]. However, these violations are rare, as we show in section VI. Therefore, a more conservative implementation could handle this process with a slow path over multiple cycles, without incurring measurable performance penalties.

A similar process occurs when a store instruction issues, when a non-zero overall VOB bit vector indicates a true vertical memory violation. This is caused by re-ordering a younger load beyond a store and thus requires a squash, which happens from the oldest load that has a non-zero VOB bit vector. The modelled microarchitecture assumes that loads are reordered with respect to earlier stores in program order based on the outcome of a store-set predictor [7]; the functionality of the predictor is largely orthogonal to SRV, and only affects vertical disambiguation, not horizontal.

### C. Horizontal disambiguation mechanism

SRV adds extra horizontal disambiguation logic alongside the conventional vertical disambiguation logic, and performs horizontal disambiguation in parallel with vertical disambiguation, as shown in red in figure 2 marked by dashed lines and titled "Horizontal Disambiguation Logic". This consists of a bit vector to identify horizontal memory-dependence violations (the *horizontal-violation bit vector*) and another to determine the bytes overlapped with the issuing load or store (the *HOB bit vector* or *horizontally overlapped bytes bit vector*).

As an illustration, in our example code from listing 3, this works as shown in figure 4. Here there are two contiguous vector accesses—the vector store from line 2, instruction A, and the second vector load, from line 6, instruction C. When instruction C issues, it sets bits in the VOB bit vector as described in section IV-B; i.e., bits 24 to 31, corresponding to the bytes that both instructions access taking into account the offset from the address-alignment base of each instruction. However, since the address-alignment offset of C is larger than that of A, there is a memory-dependence violation (a WAR violation, since the store has written bytes in later lanes before the load has read them). This is shown by the horizontal-violation bit vector, which is set from bit 24 onwards (since byte 24 is the first violating byte). The HOB bit vector is created from an AND between the VOB bit vector and the horizontal-violation bit vector, so bits 24 to 31 are set, which correctly indicate the bytes that violate the memory dependence. To deal with this violation, the vector store cannot forward these bytes to the vector load, and instead the load has to obtain all bytes from the cache.

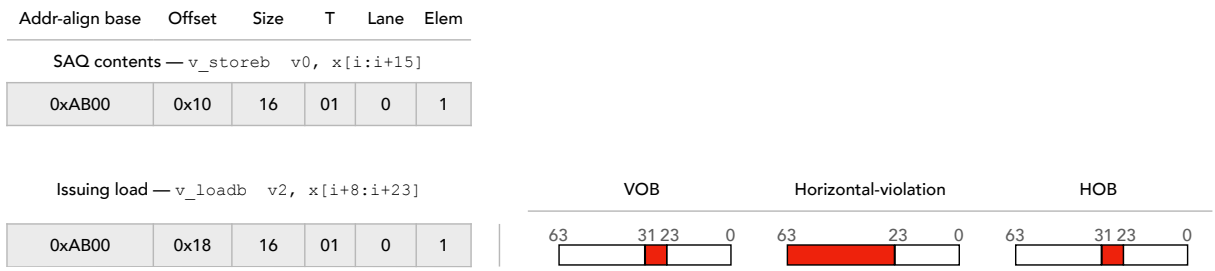


Fig. 4: Horizontal disambiguation for the code in listing 3 when the vector load at line 6 issues. Red blocks in the bit vectors indicate bits set to 1.

The example code gives an indication of how horizontal dependences can be identified for one particular pair of instructions. However, in general, identifying horizontal memory-dependence violations and data that can be forwarded from older stores to younger loads is complicated by having to deal with the various types of vector memory access that can occur (i.e., contiguous, gather/scatter, and broadcast) and the lane each instruction accesses. In the following sections, we describe how the logic works for each combination of issuing load and prior store; the same occurs for issuing stores and prior loads. For gather and scatter operations, we perform the actions for each individual load and store from the original instruction.

*1) Both contiguous:* If the address-alignment offset of the issuing load is smaller than or equal to that of a previous store in the SAQ, the overlapped bytes of the SDQ, indicated by the VOB bit vector are forwardable. The VOB bit vector is copied to the HOB bit vector. Otherwise, a horizontal WAR violation occurs and the issuing load needs to obtain forwardable data from older stores.

*2) Gather-scatter:* Here, new *lane* fields for the memory accesses are compared. If the load's lane is larger than or equal to a previous store's lane, the VOB bit vector indicates the forwardable bytes from this SDQ entry and the VOB bit vector is, again, copied to the corresponding HOB bit vector. Otherwise, a horizontal WAR violation is detected and the issuing load ignores the overlapped bytes of this SDQ entry. If there is no data forwardable from the SDQ, it will be loaded from the memory.

*3) Contiguous-scatter / gather-contiguous:* In both these scenarios each horizontal-violation bit vector is initialised by setting bits corresponding to the overlapped lanes. For a contiguous load and prior scatter the $m^{\text{th}}$ to the $(n-1)^{\text{th}}$ lanes are marked, where $m$ comes from the load's lane field and $n$ from the store's. In contrast, for a gather and contiguous prior store, all bits from the $m^{th}$ lane are marked. The bits are then shifted so that the horizontal-violation bit vector marks the bytes of data in the corresponding SDQ entry that will cause horizontal WAR violations if the issuing load accesses them. Each VOB bit vector is ORed with its corresponding horizontal-violation bit vector to indicate forwardable bytes from the SDQ entries in the HOB bit vectors.

*4) Broadcast-contiguous / broadcast-scatter:* These scenarios are variants on the situations where both instructions are contiguous or you have a contiguous load with a scatter store. The key difference is to treat the broadcast as an access to the same memory address by each lane, and construct the VOB bit vector and HOB bit vector accordingly.

### D. Execution example

We use listing 2 to provide a worked example of horizontal RAW-violation detection. We assume that the array `a` is allocated at address 0xFF00 in an LSU with an address-alignment region size of 64 bytes. The first iteration of execution is shown in figure 5. Information about the v_load instruction is recorded in an LQ entry at the top.

Within the pipeline, the scatter instruction at line 5 is split into sixteen different stores, each one 4-bytes long (integer-sized) and each taking an entry in the SAQ (and corresponding entry in the SDQ). When the scatter instruction writes to element 3 of the `a` array (step 1), it compares its address with the v_load entry in the LQ. Since the scatter writes data to `a[3]` at address 0xFF0C, it has the same address-alignment base as the v_load instruction (i.e., 0xFF00), and matches on this entry, indicating an overlapped access between the scatter and the v_load. The offsets of the scatter and the v_load instructions indicate that the overlap happens from the $12^{\text{th}}$ to the $15^{\text{th}}$ bytes, counting from 0. Therefore, the $12^{\text{th}}$ to the $15^{\text{th}}$ bits of the VOB bit vector are set to 1. All but the first 4 bits of the horizontal-violation bit vector are set to 1, since the horizontal disambiguation logic is only interested in lanes that are older than the lane of the current scatter instruction. ANDing the VOB bit vector and the horizontal-violation bit vector gives the HOB bit vector, as shown in the figure, with the $12^{\text{th}}$ to the $15^{\text{th}}$ bits set.

Execution of the scatter continues in step 2 with the part that writes to element 0 of the `a` array (i.e., `a[0]`). There is a match on the address-alignment base of the v_load again, and the VOB bit vector gets its first four bits set only. Once more, the horizontal-violation bit vector is concerned with identifying lanes from the v_load that are older than that of the current part of the scatter, which is lanes 2 onward, so all bits from the $8^{\text{th}}$ inwards are set. However, the AND between these two bit vectors, to give the HOB bit vector, produces a

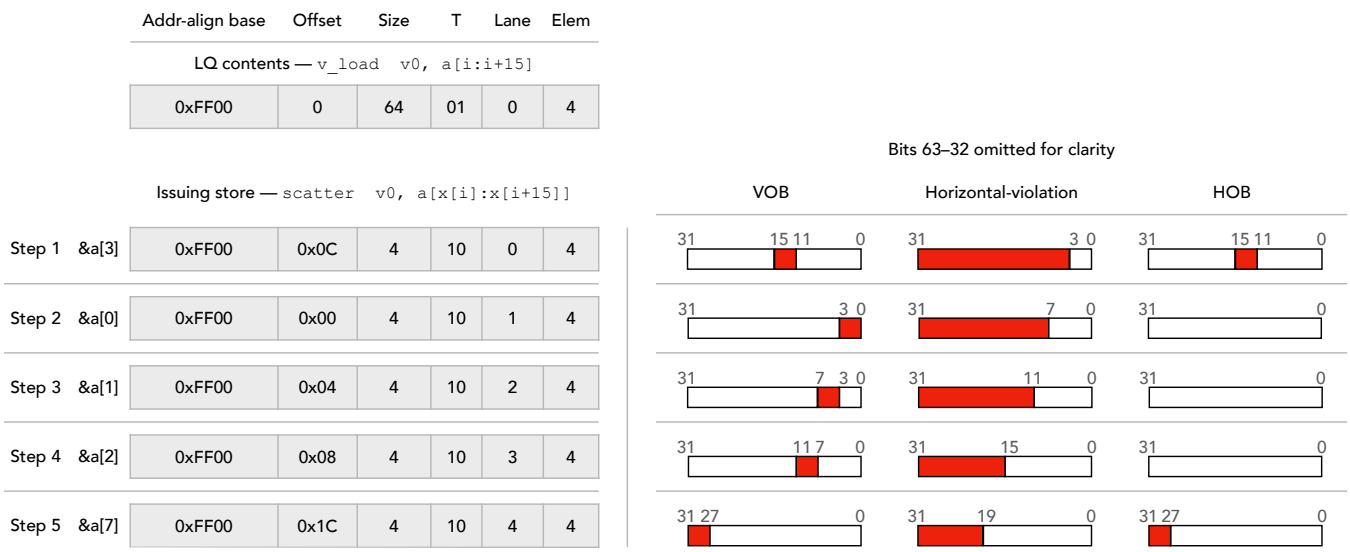


Fig. 5: Example of horizontal disambiguation when the load queue is searched during the first iteration of the code in listing 2.

bit vector with no bits set, meaning that although there is a conflict, no dependence violations occur (because data is read from and written to memory in the correct order). Continuing in this way, steps 3 and 4 (writes to array elements 1 and 2) again produce no memory-dependence violations, but the write to `a[7]` in step 5 once more produces a non-zero HOB bit vector due to the RAW violation. In this example, stores from lanes 0, 4, 8, and 12 (i.e., writes to `a[3]`, `a[7]`, `a[11]`, and `a[15]`) produce non-zero HOB bit vectors.

After all parts of the scatter instruction have executed, all HOB bit vectors are ORed together to produce the overall HOB bit vector, with bits 12–15, 28–31, 44–47, and 60–63 set. Reducing its size, based on the element size recorded in the LSU (in this case, 4 bytes), gives the SRV-needs-replay register that is consulted at the end of the SRV-region and, since it is non-zero, causes replay of lanes 3, 7, 11, and 15 so that the v_load can pick up the correct data.

## V. Experimental setup

We evaluated the performance of SRV using benchmarks taken from SPEC CPU2006 [9], representing general-purpose workloads, and applications from the NAS parallel benchmark (NPB) suite [3], Livermore [16], SSCA2 [11], the HPC Challenge benchmark suite [17] and the Rodinia benchmark suite [6], representing the HPC and scientific domains. We ran the SPEC benchmarks using reference inputs and NPB using class A inputs, taking only eleven C/C++ benchmarks from SPEC because others did not compile using our toolchain.

We modelled the SRV architecture within the gem5 simulator [5], extending a version that already contains Arm's Scalable Vector Extensions (SVE). We ran the AArch64 ISA using an out-of-order core model, fixing the vector length to 16 elements (agnostic of the element size) for all simulations. Table I describes our simulated processor.

| Parameter | Configuration |
|---|---|
| Core | Out-of-order, 3GHz |
| Pipeline | Fetch / decode / issue width: 8 |
| LSU | 64-entry |
| IQ | 32-entry |
| ROB | 400-entry |
| Ports | SAQ: 6 (2 reads, 2 writes, 2 CAMs)<br>SDQ: 7 (5 reads, 2 writes)<br>Vec Reg File: 8 (6 reads, 2 writes)<br>Cache: 2 (1 read / write, 1 read-only) |
| Vec-op / cycle | Non-mem: 2 integers, 1 others<br>Mem: 2 loads, 1 store |
| Branch pred | 64-entry local, 1024-entry global, 128-entry-BTB, 1024-entry chooser, 8-entry RAS |
| L1 cache | 32KiB, 4-way, 2-cycle hit lat |
| L2 cache | 1MiB, 16-way, 7-cycle hit lat |

TABLE I: Core and memory experimental setup.

To compile our applications we first extended a version of LLVM that performs auto-vectorisation with SVE. Although the application of SRV is not limited to loops only, we currently only focus on loop-level vectorisation. Within the compiler, we modified the analysis passes to enable the compiler to identify loops that have statically unknown memory dependencies. We then allowed the compiler to bypass the memory-safety check for these loops using the same mechanism that an OpenMP hint does, so that a later transformation stage performs vectorisation regardless of the memory dependences. Based on this transformation, we amended the pass to bound these loops within SRV-start and SRV-end instructions. All workloads were then compiled with optimisation level -O3 to create the baseline, SVE and SRV binaries.

We simulated SRV-vectorisable loops in the detailed out-of-order CPU model, fast-forwarding through other code but maintaining a dynamic instruction count. Due to prohibitively

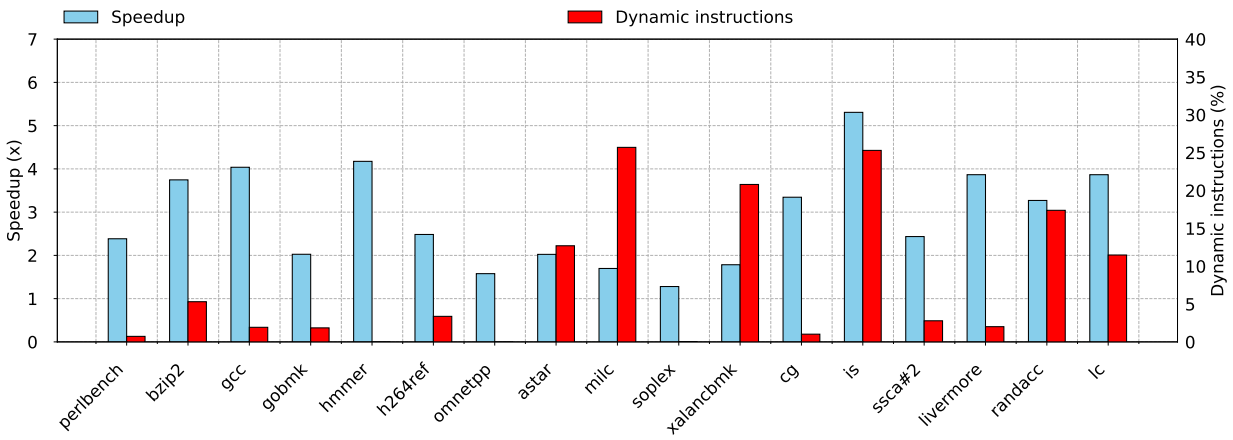


Fig. 6: Per-loop speedup for all SRV-vectorisable loops in each benchmark and their corresponding coverage in dynamic instructions compared to a baseline out-of-order microarchitecture.

long simulation times for SPEC CPU2006 benchmarks, only the first invocation of SRV-vectorisable loops were simulated in the detailed out-of-order CPU model. These loops have a range of sizes, from 1.3K to 200M dynamic instructions, with an average of 94.9M dynamic instructions per loop across all workloads, in the invocations that we simulate. Experiments we performed on instrumented versions of the SPEC benchmarks that track memory dependences have shown that there is less than 1% difference in results when running all invocations compared with just running the first invocation of each loop. Based on this, we conclude that, in SPEC benchmarks, later invocations of a loop have the same behaviour and characteristics as the first one. All invocations of SRV-vectorisable loops from other benchmarks were simulated in full.

## VI. Performance Evaluation

Figure 6 presents the speedup of all SRV-vectorisable loops (i.e., those that cannot be vectorised without SRV). The results are normalised to the performance when vectorising with SVE, which has its performance on a wide range of benchmarks reported by Stephens et al. [23]. In general, loops achieve a speedup of 2× or more. Loops in *bzip2*, *gcc*, *hmmer*, *livermore* and *lc* achieve a loop speedup close to 4× and *is* achieves a loop of speedup of over 5×. This is because the majority of the instructions in SRV-vectorisable loops are already vectorisable using existing techniques, but the compiler's imprecise alias analysis hinders vectorisation. However, *omnetpp*, *soplex* and *xalancbmk* achieve relatively low loop speedups (1.49×, 1.29× and 1.78×) since some of the SRV-vectorisable loops have high memory-to-computation ratios in which one operation requires multiple gather instructions to prepare data for it.

Most benchmarks have SRV-vectorisable loops that cover less than 5% of the total dynamic instructions. The percentage is low since SRV only vectorises loops with unknown memory dependences as the sole feature that prevents vectorisation. Further enhancing the coverage of vectorisation requires more advanced compiler techniques to be used together with SRV. However, *astar*, *milc*, *xalancbmk*, *is*, *randacc*, and *lc* have a considerable amount of code that can take advantage of SRV, accounting for 12.7%, 25.7%, 20.8%, 25.3%, 17.3% and 11.4% of the total dynamic instruction count respectively.

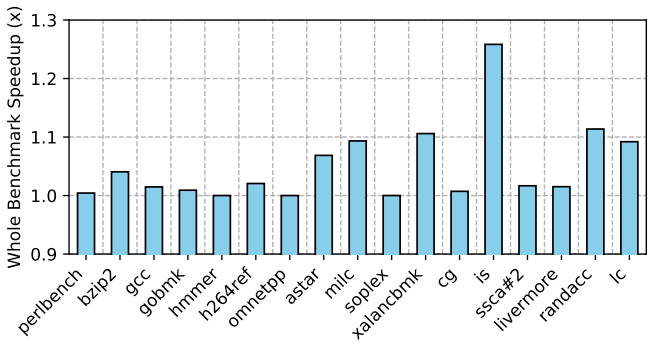


Fig. 7: Whole-program speedup for each benchmark compared to vectorisation with SVE.

Figure 7 presents the whole-program speedup of each benchmark, calculated based on the dynamic instruction count of the SRV-vectorisable loops and their coverage. Once again, these results represent the additional benefit that SRV brings beyond state-of-the-art vectorisation. These show speedups of up to 1.09× for SPEC benchmarks and 1.19× for other applications (geometric mean 1.04× and 1.10×) over SVE vectorisation.

SRV achieves speedups of up to 1.1× for SPEC benchmarks and up to 1.26× for other applications. Floating-point benchmarks in SPEC benefit more from SRV since SRV-vectorisable loops in these benchmarks cover a larger percentage of dynamic instructions. *bzip2*, *gcc*, *gobmk* and *h264ref* also produce observable speedups (more than 1%) when more loops are vectorised by SRV. The other SPEC benchmarks receive negligible performance enhancement since

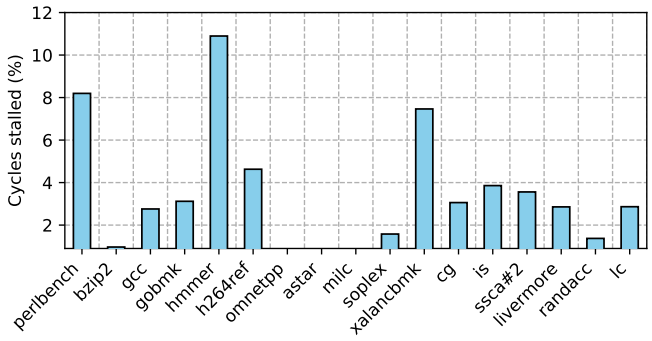


Fig. 8: Fraction of execution barrier cycles in SRV-vectorised loops.

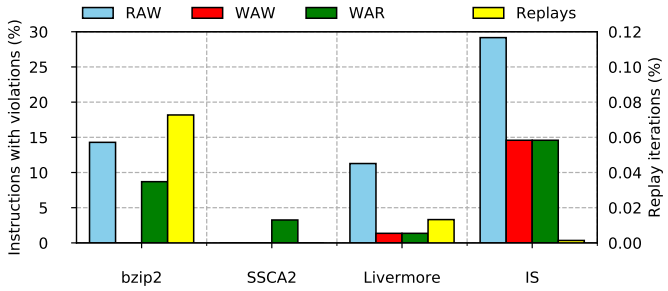


Fig. 9: Percentage of different violations and re-execution in SRV-vectorised loops.

SRV only covers a very small fraction of the whole program. *is* gives the most obvious speedup since loops that can be vectorised by SRV cover a considerable fraction of the whole program. Moreover, the loop that covers the biggest fraction of *is* has all but one operation vectorisable using existing techniques. SRV enables vectorisation of this operation, thus realising a speedup of 1.26×. Overall, the geometric mean speedup is 1.05×.

### A. Overhead analysis

SRV introduces two types of performance overhead. The first is shown in figure 8, which is the number of cycles introduced by execution barriers as a fraction of the total cycles within all SRV-vectorisable loops of each benchmark. In other words, this is the number of cycles each SRV-end instruction stalls the issue of later instructions until it has executed due to serialisation (see section III-D1). Most benchmarks have execution barriers that are less than 4% of total execution cycles of SRV-vectorisable loops. Among them, *bzip2*, *omnetpp*, *astar*, and *milc* have negligible performance overhead (0.9%, 0.03%, 0.12%, and 0.05% respectively) caused by execution barriers. However, execution barriers are more significant in *perlbench*, *hmmer*, *h264ref*, and *xalancbmk*. This is because some SRV-vectorised loops in these benchmarks are small with short trip counts.

Another source of overhead comes from the occurrence of memory-dependence violations that require rollback and

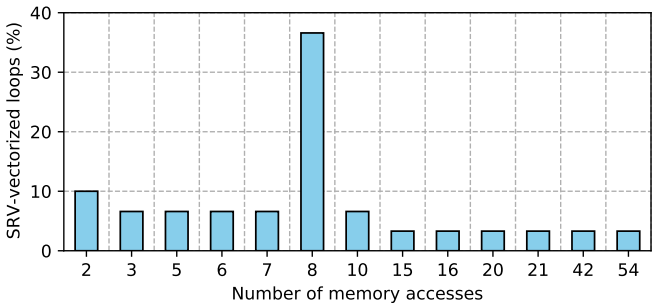


Fig. 10: SRV-vectorised loops, broken down by their number of memory accesses.

replay. Figure 9 shows the number of memory-dependence violations of each type as a fraction of the total instructions within our vectorised loops. We only show the four benchmarks that actually incur violations at run time. The others contain loops that the static analysis suggests may have violations, but at run time none actually occur.

Each benchmark has four bars. The first three represent the number of different types of memory-dependence violation per static loop instruction. The fourth bar shows the overhead of replays as a fraction of the number of vector iterations. In all cases, lower is better.

RAW dependences cause the most violations. In *bzip2*, 14% of the instructions cause RAW dependence violations. These require replay, meaning 0.07% additional vector iterations. In contrast *is* has 29% of its instructions causing RAW dependence violations but this translates into an overhead of only 0.001% iterations. This is because there are few instructions in these loops, making each instruction with a violation account for a large percentage in figure 9.

A key advantage of SRV is to provide a guarantee of correct execution. In the loops we vectorise, the compiler cannot prove there are no cross-iteration dependences due to their dynamic nature. Even though, as figure 9 shows, conflicts are rare, they can happen in the middle of a vector instruction and need handling. Furthermore, since the compiler cannot prove their absence in any of the loops we vectorise, *none* of these loops are profitably vectorisable using existing hardware and compiler support.

### B. Hardware parameters

Figure 10 shows the number of memory accesses in loops that were previously unvectorised due to statically undecidable dependences, but which SRV can vectorise safely. The majority of loops (80%) have ten memory accesses or fewer, although a few have more than 16 memory-access instructions. Large numbers of accesses occur due to loading from or storing to multi-dimensional arrays, or pointer chasing, for example. All loops with ten memory accesses, or fewer, contain a maximum of three gather-scatter instructions; in fact, in our workloads, only 5.8% of loads are gathers. We break these into multiple micro-ops, and each accesses the LSU

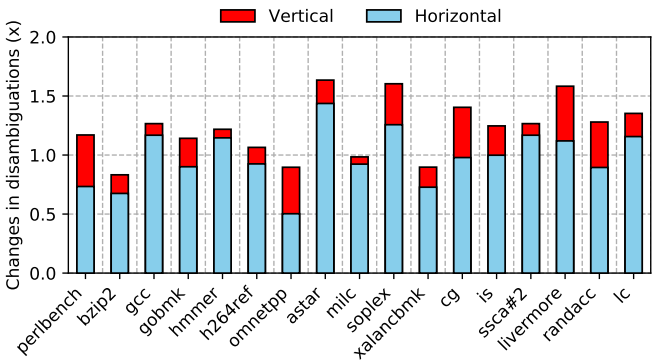


Fig. 11: Changes in the number of address disambiguations in SRV-vectorised loops.

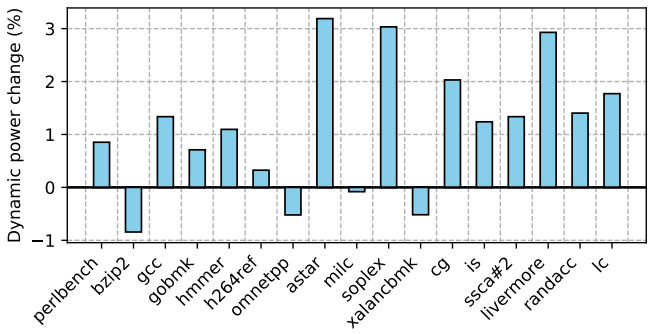


Fig. 12: Changes in dynamic power consumption introduced by SRV.

independently over a number of cycles, which is the common method of implementation across commercial designs. Since the simulation was operated on 16-element vectors with 64 LSU entries, SRV contains enough entries in its LSU to execute these loops without overflow $(16*3+(10-3)=55)$.

SRV enhances the coverage of vectorisation at the cost of additional horizontal address disambiguation. Vectorisation reduces the number of dynamic instructions and, therefore, potentially reduces the number of address disambiguations required. However, SRV introduces extra horizontal address disambiguations for each memory-access instruction at every vector iteration. Figure 11 presents the number of address disambiguations when executing loops vectorised through SRV compared sequential execution, broken down by type. As can be seen, SRV increases the number of address disambiguations by up to 60%. For benchmarks including *bzip2*, *omnetpp*, *milc* and *xalancbmk*, SRV incurs fewer address disambiguations compared to sequential execution, since vectorisation reduces the number of dynamic instructions.

Among the total number of address disambiguations, the horizontal ones take up a large fraction. This is because, in the SRV-region, horizontal disambiguations replace vertical ones when executing load instructions, while both horizontal and vertical disambiguations occur when executing store instructions. Although horizontal disambiguations incur more comparisons and bit-vector shifts, the SRV hardware is only powered on when executing SRV-vectorisable loops. Hence the power overhead of SRV is almost negligible. as we explore in the next section.

### C. Power analysis

We modified McPAT [15] to analyse the power introduced by SRV. McPAT models the power of CAM lookup operations to calculate the dynamic power consumption of the processor. An out-of-order issue of a load instruction is modelled by one CAM lookup to the store buffer (for store-to-load forwarding) and one CAM lookup to the load buffer to maintain the order of load instructions [15]. For an out-of-order issue of a store instruction, one CAM lookup to the load buffer is modelled to find younger loads that must be squashed.

To model SRV, we doubled the CAM lookup operations and added an extra CAM lookup to the store buffer for store instructions, in the SRV-region only, to reflect the extra CAM lookup caused by the horizontal disambiguation process. CAM lookups for load instructions are unchanged since horizontal disambiguation replaces vertical disambiguation when executing load instructions in the SRV-region. However, given the limited detail McPAT can model, the extra bit-vector shifts incurred in horizontal disambiguation are not modelled.

Figure 12 compares the run-time power of the core when running benchmarks using SRV and without vectorisation. In general, more CAM lookups cause more power consumption. However, when considering the run-time power of the whole core, the changes are negligible. This is because the LSU only contributes 11% of the total run-time power on average across the tested benchmarks. For benchmarks including *bzip2*, *omnetpp*, *milc* and *xalancbmk*, SRV results in lower power consumption due to a reduction in address disambiguation. For other benchmarks, SRV adds no more than 3.2% to the total run-time power.

### D. Comparison with FlexVec

The closest technique to SRV is FlexVec [2], which adds compiler-generated run-time checks for memory-dependence violations and limits the vector width to the number of lanes that do not incur any violation. SRV differs from FlexVec by using an implicit memory-disambiguation mechanism, which allows lanes after the first violating lane to execute. This means that, in cases where there are several violations, SRV can potentially execute a loop with fewer iterations than FlexVec, due to its selective replay of only those lanes with violations, and vectorise a loop more efficiently due to its implicit memory-disambiguation mechanism.

Unfortunately we are not aware of any compiler that currently implements FlexVec and the authors did not open-source their tools. We therefore compare against FlexVec by faithfully reproducing their technique for dealing with loops whose memory-dependence information is unknown at compile time. We model FlexVec and SRV in an emulator that was validated against our gem5 implementation of SRV.

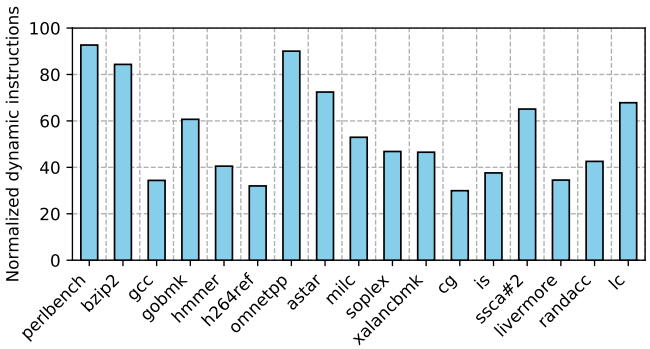


Fig. 13: Dynamic instruction count required by SRV compared to those required by FlexVec.

When emulating, we broke the VCONFLICTM instruction into several instructions, with each one comparing one element of a source vector with all enabled previous elements of a target vector, to account for FlexVec's address-disambiguation mechanism. Partial vectorisation was already modelled for SRV. FlexVec's evaluation only executed samples of code surrounding and including vectorised loops that have been shown to be profitable through use of a profiler. In contrast, we execute all loops that are vectorisable, as described in section V, to show the full impact of vectorisation.

Figure 13 compares the dynamic instruction count of target loops for both techniques. This shows that SRV requires fewer than 60% dynamic instructions to vectorise loops, compared with FlexVec, for most benchmarks. This is mainly because SRV does not incur extra instructions to perform run-time checks. Moreover, fewer vector iterations required by SRV also reduce the required dynamic instruction count.

### E. Summary

We have presented the performance improvement introduced by SRV, and analysed SRV's performance overheads before comparing it with FlexVec, the closest existing technique from the literature. The SRV architecture demonstrates that judicious use of speculation can unleash data-level parallelism performance and bring the benefits of SIMD execution to a wider range of codes.

## VII. Related work

Speculative vectorisation techniques have been proposed for both compilers and architectures. Sujon et al. [24] vectorise loops when the source of a cross-iteration dependence is guarded by a conditional statement that is rarely true. Code is generated to check the condition for all lanes of a vector iteration in advance. The vector code is only executed if the condition is false for all vector lanes, otherwise the fallback scalar code is executed. Application of such speculative vectorisation is limited as the conditional statement itself cannot be part of any cross-iteration dependence cycle. Moreover, many partially vectorisable loops are not vectorised using this technique.

Another approach uses a combination of an inspector and executor to parallelise loops [22]. The compiler generates inspector code that analyses cross-iteration dependences within the loop at run time. An executor program then employs specific optimisations to the loop iterations using the dependence information produced by the inspector. Techniques such as this are often associated with high overhead since they generally require large additional data structures and extra memory operations.

Baghsorkhi et al. [2] proposed FlexVec, which partially vectorises loops by generating code to perform run-time alias checking. However, the overhead of this technique is highly dependent on the frequency of the pointer aliases or memory-dependence violations. Moreover, FlexVec does not attempt to vectorise any further once a pointer alias or memory dependence is detected. This reduces the effectiveness of wide vector registers, leaving them underutilised.

On the architecture side, hardware approaches [13, 21] are proposed in order to detect pointer aliases or memory-dependence violations. However, all these prior works focus on violation detection and, once a violation has been detected, execution rolls back to sequential mode. Therefore, application of such techniques is limited and the potential for exploiting data-level parallelism is not realised.

## VIII. Conclusion and future work

We have presented selective-replay vectorisation, a technique to safely vectorise code with periodic or unknown memory data dependences. Our hardware monitors memory addresses accessed within a compiler-annotated loop and replays only those SIMD lanes that experience memory-dependence violations. Evaluation shows loop speedups of up to 5.3× and whole-program speedups of up to 1.19× across a range of general-purpose and HPC applications over already-vectorised code. In conclusion, our SRV architecture demonstrates that judicious use of speculation can unleash data-level parallelism performance and bring the benefits of SIMD execution to a wider range of code. Future work will consider advanced compiler techniques that take advantage of SRV to further enhance the coverage of vectorisation, and develop optimisations, such as removing the serialisation barrier in SRV-end, to improve performance and power efficiency.

## Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1 and EP/P020011/1, and Arm Ltd. Additional data related to this publication is available in the data repository at https://doi.org/10.17863/CAM.68206.

## References

[1] Arm. Arm architecture reference manual. https://developer.arm.com, 2019.

[2] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu. FlexVec: Auto-vectorization for irregular loops. In *PLDI*, 2016.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks—summary and preliminary results. In *Supercomputing*, 1991.

[4] U. Banerjee. *Speedup of Ordinary Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1979.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), 2011.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[7] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *ISCA*, 1998.

[8] B. Guo, Y. Wu, C. Wang, M. J. Bridges, G. Ottoni, N. Vachharajani, J. Chang, and D. I. August. Selective runtime memory disambiguation in a dynamic binary translator. In *CC*, 2006.

[9] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4), 2006.

[10] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan. Dynamic trace-based analysis of vectorization potential of applications. In *PLDI*, 2012.

[11] HPC. HPC graph analysis. http://www.graphanalysis.org/benchmark, 2017.

[12] Intel. Intel advanced vector extension 512 (Intel AVX512). https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/avx-512-overview.html, 2018.

[13] R. Kumar, A. Martínez, and A. González. Assisting static compiler vectorization with a speculative dynamic vectorizer in an HW/SW codesigned environment. *ACM Trans. Comput. Syst.*, 33(4), 2016.

[14] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, 2000.

[15] 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 *MICRO*, 2009.

[16] Livermore. Livermore loops coded in C. http://www.netlib.org/benchmark/livermorec, 1992.

[17] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The HPC challenge (HPCC) benchmark suite. In *Supercomputing*, 2006.

[18] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua. An evaluation of vectorizing compilers. In *PACT*, 2011.

[19] A. Nicolau. *Parallelism, Memory Anti-aliasing and Correctness for Trace Scheduling Compilers (Disambiguation, Flow-analysis, Compaction)*. PhD thesis, Yale University, 1984.

[20] M. Ohmacht, A. Wang, T. Gooding, B. Nathanson, I. Nair, G. Janssen, M. Schaal, and B. Steinmacher-Burow. IBM Blue Gene/Q memory subsystem with speculative execution and transactional memory. *IBM J. Res. Dev.*, 2013.

[21] A. Pajuelo, A. González, and M. Valero. Speculative dynamic vectorization. In *ISCA*, 2002.

[22] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *J. Parallel Distrib. Comput.*, 8(4), 1990.

[23] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. The ARM scalable vector extension. *IEEE Micro*, 37(2), 2017.

[24] M. H. Sujon, R. C. Whaley, and Q. Yi. Vectorization past dependent branches through speculation. In *PACT*, 2013.

[25] VPlan. Vectorization plan. https://llvm.org/docs/Proposals/VectorizationPlan.html, 2018.

[26] D. Witt. System for store to load forwarding of individual bytes from separate store buffer entries to form a single load word. United States patent number 6141747, 2000.

[27] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, 1982.

[28] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *HPCSA*, 2013.