Scalar Vector Runahead

Jaime Roelandts[†], Ajeya Naithani[†], Sam Ainsworth[≀], Timothy M. Jones[‡] and Lieven Eeckhout[†]

[†]Ghent University, Belgium [≀]University of Edinburgh, UK [‡]University of Cambridge, UK

***Abstract*—Modern graph and database processing typically takes place on high-end servers in data centers. However, with growing concerns of data privacy, trustworthiness, and all-time connectivity, there has been a shift toward increased analytics processing on edge devices such as mobile phones. In an ideal scenario, we would run these applications on the energy-efficient in-order cores available in these systems rather than the power-hungry out-of-order cores. However, these applications typically feature an extremely low computation-to-communication ratio and irregular memory accesses, meaning their performance is memory-bound, and out-of-order cores provide significant performance advantages over their in-order counterparts. Although prior work on Vector Runahead has substantially improved the performance of graph applications on very large out-of-order cores, it incurs high complexity and power consumption, so is unsuitable for energy-efficient in-order processors.**

Scalar Vector Runahead (SVR) extracts high memory-level parallelism on simple in-order cores by piggybacking on existing instructions executed on the processor leading to future irregular memory accesses. SVR executes multiple transient, independent, parallel instances of memory accesses and their chains initiated from different values of a predicted induction variable to move mutually independent memory accesses next to each other to hide dependent stalls. With a hardware overhead of only 2 KiB, SVR delivers 3.2× higher performance than a baseline 3-wide in-order core, and 1.3× higher performance than a full out-of-order core, while halving energy consumption. Increasing the overhead to 9 KiB to account for a larger register file, SVR can extend the speedup relative to an out-of-order core to 1.7×.

***Index Terms*—microarchitecture (CPU), data prefetching, runahead, graph processing**

I. Introduction

The importance of workloads with chains of dependent memory accesses, as seen in graph analytics, database and high-performance computing (HPC), has consistently increased over the past few years, which has resulted in significant research efforts to improve their performance [1], [3], [31], [40], [43], [53]. To execute these workloads efficiently, the goal is to overlap as many of their cache misses as possible to hide their memory access latency. The traditional method of overlapping execution relies on out-of-order (OoO) superscalar processors, which use a reorder buffer to find independent work. However, even the largest out-of-order systems struggle to find enough independent work to achieve good performance on these challenging workloads [43]. At the same time, out-of-order processors are widely known to be power-hungry [20], [52], and as a result, today's edge processors also deploy in-order superscalar cores to reduce energy consumption [21].

Despite edge analytics being increasingly important due to privacy and latency concerns [45], these in-order cores struggle immensely with big-data workloads and the indirect memory accesses seen within. In-order cores are unable to overlap memory accesses by dynamically rescheduling them, so have no ability to hide the associated memory latencies: we see cycles-per-instruction (CPI) as high as 20, and so energy efficiency can be even lower than for out-of-order cores.

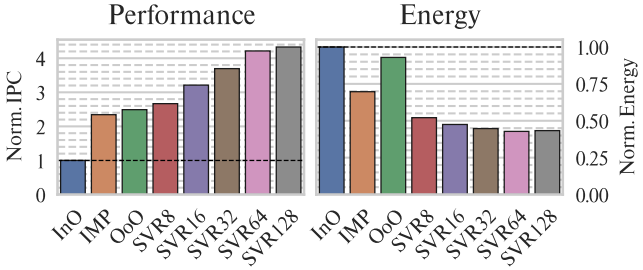


Fig. 1: Average speedup (harmonic mean of IPC) and whole-system energy for various lengths of SVR (16 default) versus an in-order baseline, an out-of-order core, and IMP [60].

Can we achieve high performance for these challenging workloads with chains of dependent memory accesses, while still reaping the energy-efficiency benefits of in-order processing? To answer this question, we take inspiration from the state-of-the-art in data prefetching on large out-of-order superscalar microarchitectures, namely Vector Runahead [40], [43]. The key motivation behind such techniques is that, by using a stride-load predictor to predict some future memory accesses, we can spawn many independent chains of dependent instructions to find many other future memory accesses, thus using transient execution to prefetch data for standard out-of-order execution. However, the state-of-the-art technique, Decoupled Vector Runahead [43], makes heavy use of resources only available on large cores: the runahead mode has its own fully decoupled, dedicated execution context within the core, that can overlap with the main thread. This context is small relative to an out-of-order core but would be infeasibly large next to an in-order core. It also requires very wide vector units unavailable on small cores, a very large physical register file to store all the overlapping operations, and out-of-order execution to efficiently run a 'discovery pass' for loop bounds.

Our aim is to take the principles of Vector Runahead and apply them to small in-order superscalar cores. This requires rearchitecting the approach, keeping the notion of overlapped future memory accesses to exploit MLP, but avoiding the need for a supported vector ISA or a separate runahead context. Our technique, called *Scalar Vector Runahead (SVR)*, synchronously 'piggybacks' off existing instructions executed in

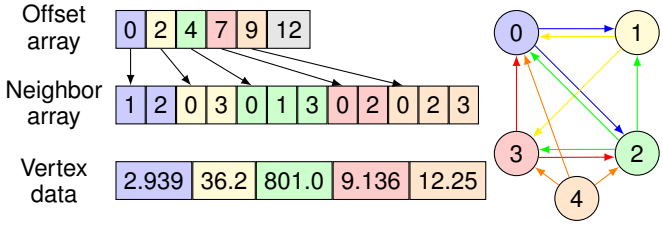


Fig. 2: A sample graph and its representation in CSR format. Both the offset and neighbor arrays store values in a contiguous manner. The index of each element in the offset array represents vertex ID while the value represents the index of its first neighbor. Values in the neighbor array represent the index of the elements in the vertex data array.

```
1  for (NodeID u=0; u < g.num_nodes(); u++) {
2    ScoreT incoming_total = 0;
3    for (NodeID v : g.in_neigh(u))
4      incoming_total += outgoing_contrib[v];
5  }
```

Listing 1: C++ code for the hot loop in PageRank.

the processor, by opportunistically generating multiple scalar copies of them, each with independent sets of inputs and dependencies. It keeps track of all the newly interleaved instructions with their own small pool of SRAM, while learning from dynamic properties of execution to efficiently discover accurate loop bounds and without relying on out-of-order execution to calculate properties of future work. It does this while retaining a simple, strictly in-order execution model.

Compared to a 3-wide in-order core configured after an ARM Cortex-A510, SVR delivers 3.2× higher performance across a variety of memory-latency-bound graph, database and HPC workloads with its default vector length of 16 and only 2 KiB state. This surpasses even a 3-wide out-of-order (OoO) core by 1.3×, rising to 4.3× and 1.7× higher than an in-order and OoO core, respectively, for SVR with a 128 vector length and 9 KiB state. SVR requires 53% and 49% lower energy than in-order and out-of-order cores, respectively (Figure 1). In addition, SVR beats the Indirect Memory Prefetcher (IMP) [60] by 1.4×. SVR establishes these speedups without dedicated vector execution units.

II. Background

A. Sparse Memory Accesses

Many classes of modern workloads feature difficult to predict, non-sequential memory accesses, based on indexed lookups, e.g., to sparse matrices (in graph and HPC) or table data structures (such as the hash tables seen in database workloads). While these workloads theoretically have high *memory-level parallelism (MLP)* at an algorithmic level, this is difficult to exploit. Large data volumes and low temporal locality leave modern cache hierarchies underutilized. Because the accesses are indirect or pointer-chasing, they cannot be easily prefetched closer to the core in the cache hierarchy.

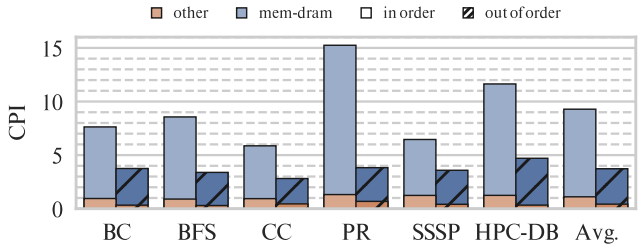


Fig. 3: CPI stacks for workloads with chains of dependent memory accesses on in-order versus OoO cores. *In-order cores spend 2.5× more cycles waiting for DRAM than OoO cores.*

As an example, due to their sparse nature, graph applications are typically stored a compressed sparse row (CSR) format for high storage efficiency. As shown in Figure 2, this consists of three arrays: offset, neighbor, and vertex data [35]. Accesses to the offset array are sequential while accesses to the neighbor and vertex data arrays are indirect. Listing 1 shows C++ code accessing the three arrays for PageRank from the GAP benchmark suite [11]. Line 1 iterates over the nodes, which is represented by the Offset array in Figure 2, line 3 iterates over the neighbors of each node, which is represented by the Neighbor array in Figure 2, and line 4 adds the contribution of each neighbor to the current node ranking, which is an access to the Vertex data in Figure 2.

Accesses to `g.in_neigh(u)` follow a regular sequence, so they can be picked up by a stride prefetcher, whereas accesses to `outgoing_contrib[v]`, based on the non-sequential values stored inside `g.in_neigh(u)`, are irregular and highly data-dependent, meaning they persistently miss in the cache. Although an out-of-order core can accommodate a few loop iterations simultaneously in its reorder buffer to extract MLP from multiple dependent chains, it quickly stalls before it can saturate memory bandwidth. For in-order cores, the problem is even worse. Even if we assume that the in-order core is stall-on-use rather than stall-on-load, the use of the cache-missing load will cause the core to pause execution until the high-latency access to main memory completes. This effectively inhibits any kind of MLP, causing poor performance.

B. Memory Latency on In-Order Cores

Figure 3 reports the cycles-per-instruction (CPI) for in-order and OoO cores. As the figure demonstrates, the severity of the performance impact of indirect memory accesses on an in-order core is manifold higher than its OoO counterpart. While the OoO core is stalled for 3.6 CPI on DRAM memory accesses, the in-order core stalls for a substantially higher 8.9 CPI. This results in a 2.5× slowdown on the in-order relative to the out-of-order core. Therefore, addressing the performance bottleneck for applications with indirect memory accesses is critical on in-order cores.

TABLE I: Differences between VR, DVR and SVR.

	VR	DVR	SVR
Based on existing vector ISAs	Y	Y	N
Relies on existing vector registers	Y	Y	N
Optimizes vector-register usage	N	N	Y
Stalls the main thread	Y	N	N
Runahead synchronous with main thread	N	N	Y
Mitigates incorrect prefetches	N	Y	Y
Needs a discovery pass	N	Y	N

C. Vector Runahead

Vector Runahead (VR) [38], [40] is a microarchitectural technique for out-of-order cores, to generate high MLP for sparse memory accesses. VR keeps a reference prediction table [17] to find loads with striding access patterns, which it uses to predict induction variables for future loops. If these stride loads generate dependent memory accesses down their compute chain, VR creates instructions corresponding to chains beginning from many different indices, transiently issuing many independent future iterations simultaneously. VR combines these multiple scalar-instruction copies into vectors to improve back-end throughput. Though the dependent accesses will still miss in the cache, many independent misses now occur in close proximity, resulting in very high MLP, and a high cache hit rate once the processor returns to normal execution.

Decoupled Vector Runahead (DVR) [39], [43] builds on top of VR. While VR is triggered when the ROB fills up and so the OoO core cannot progress further, DVR is 'decoupled' or independent of the size of the microarchitectural structures by issuing a speculatively vectorized instruction stream in a second, simple in-order simultaneous subthread on the same core. In addition, DVR detects loop bounds at run time to improve prefetching accuracy, including across multiple nested inner- and outer-loops simultaneously. DVR also handles control-flow divergence using GPU-style reconvergence stacks.

Both techniques make extensive use of resources only available or realistic in large out-of-order superscalar cores, meaning a direct translation to energy-efficient in-order cores is impossible. However, both also give insights into designing mechanisms that can obviate the even more pressing latency-bound behavior observed on in-order cores.

III. SVR: Design Philosophy

While in-order cores are widely deployed, they suffer even more heavily than out-of-order cores when it comes to latency-bound irregular workloads. At the same time, the state-of-the-art solutions for such workloads, Vector Runahead [40] and Decoupled Vector Runahead [43], are completely unviable on small cores. A comparison of the guiding principles and characteristics of VR, DVR, and our new technique, SVR, are summarized in Table I and elaborated on below.

Scalar-Vectors, Not Vectors: Both VR and DVR pack data for the independent groups of future iterations into wide SIMD registers and utilize their corresponding SIMD vector instructions (such as AVX512). Smaller cores are not likely to support vector operations at all, or only to support small vector widths (e.g., 128 bits [21]). Still, even with 512-bit datapaths, VR and DVR must issue multiple vector instructions at once (VR issues 64 iterations simultaneously, and DVR up to 128). If the underlying core is memory bound, we can still achieve speedup by issuing the computation on the many independent data items as scalar operations rather than as vector instructions, hence the term '*scalar vectors*'. Even in an in-order core, provided the core supports stall-on-use rather than stall-on-miss (i.e., the core stalls when a loaded value is used rather than immediately on a cache miss), we can still take advantage of data-level parallelism to achieve high MLP by packing many independent memory accesses together. Even DVR's subthread does not make use of out-of-order execution, as the parallelism between lanes is ample, and individual lanes have little instruction-level parallelism due to indirect chains. Thus, we can see the MLP benefits of transient vectorization without actually using out-of-order execution or any explicit vector operations, hence the proposal of *Scalar* Vector Runahead.

No Spare Registers: Both VR and DVR perform transient execution by repurposing spare physical registers. In-order cores do not need to perform renaming, and so they do not have any physical registers. Indeed, VR and DVR not only need some spare registers, but lots of them: the techniques deliberately overlap the liveness of all registers in up to 128 loop iterations simultaneously to achieve high MLP. SVR therefore needs explicit new storage for storing intermediate results of the transient runahead execution, which is additional overhead that will be particularly prominent if the core is small to begin with. In order to reduce resource utilization, we need to recycle registers before we can guarantee they are dead, even at the expense of failing to generate runahead.

Lockstep Coupling: VR only triggers once the ROB is full — a structure in-order cores do not have to begin with. DVR does not have such a precondition due to running inside a second independent simultaneous subthread. The overheads of the latter on a large out-of-order core may be insignificant, but will be costly on a small in-order core. In addition, both mechanisms have distinct state for transient and non-transient execution, including checkpoints of all scalar registers, again stored in spare register file space the little core will not have.

We can avoid the complexity of both mechanisms, while building a design suitable for coupling to an in-order core, by running the runahead instructions in lockstep with real execution: for example, a stride load is immediately succeeded by multiple future scalar copies (scalar-vector instructions) of the same stride load, which are followed by the next scalar instruction, which are followed by copies of that same instruction with the references changed depending on which scalar-vector element it is associated with. This means we need only one copy of all scalar values, rather than one for runahead and one for true execution. For example, the base address of an array can be shared and stored once.

Timeliness-Accuracy Trade-Offs: The original VR is inaccurate as it always issues 64 iterations ahead even if this ends

up outside the data structure or outside the loop bounds of the original code. DVR is accurate because it dynamically discovers loop bounds: it then waits until loop conditions are evaluated at the end of the first iteration after its discovery mode begins to initiate runahead. This is suitable on an out-of-order core, where such a condition is likely to be evaluated shortly after entering the loop, but on an in-order core, it is likely to be evaluated after the long-latency loads inside the loop. This significantly hurts MLP, as it means whole iterations effectively end up completely in-order with no runahead. To solve this, SVR uses a novel combination of both history predictors and *scavenging*, where it accesses data that is likely to be representative of the future loop condition.

These fundamental principles: (1) using scalars not vectors to execute our vectorized code, (2) frugally using specially provided temporary register storage for the runahead execution, (3) coupling runahead with the main thread to avoid stalling the processor or needing an extra context, and (4) using information that is cheap for an in-order core to discover to improve prefetcher accuracy, are all vital insights that we use to build SVR in the following section.

IV. SVR Microarchitecture

SVR builds on a stall-on-use in-order core [26]. The core monitors the data-load stream for striding memory access patterns, generating multiple scalar copies of striding loads each with a different future address. Transitive dependent instructions of the striding load are tracked with the help of a taint tracker, and multiple copies are generated for each of the dependent instructions as well. The group of generated scalar copies of data for a single instruction is called a *scalar vector (SV)*, and the instruction copies themselves one collective *scalar-vector instruction (SVI)*. One SV comprises N scalars (or the vector has N elements); $N = 16$ in this work unless otherwise specified. The key difference between SVIs and vector instructions is that the scalar instructions comprising an SVI are issued and executed independently in the processor back-end. An SVI executes transiently on the existing functional units alongside the main thread (meaning they cannot affect architectural state), and SVIs' purpose is to prefetch future memory accesses for the main thread. Since there are no spare physical registers in an in-order core, SVIs read and write to a small physical vector register file called the *speculative register file (SRF)*.

Upon detecting a load instruction with a striding memory access pattern, the core enters into *piggyback runahead mode (PRM)* and generates an SVI for the striding load, see also Figure 4. SVIs piggyback on the existing real instruction stream executed by the processor, in that SVIs are also generated for any instruction that directly or indirectly depends on the striding load, at the point the real instruction is issued.

The chain of dependent instructions starting from the striding load is known as an *indirect chain*. When the core encounters another instance of the same striding load, the process of generating SVIs terminates and the core enters into *waiting mode*. This marks the execution of one iteration of

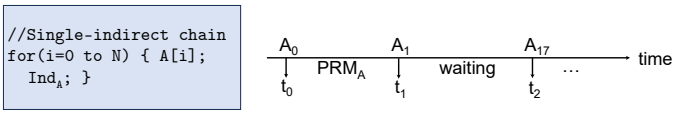


Fig. 4: Example indirect chain starting from a striding load A. At the time of loading A_0, PRM transiently overlaps the issuing of $A_{1..16}$ for prefetching, and does the same with its dependents Ind_A. When we reach A_1 in true execution we enter waiting mode, where we reap the rewards of earlier prefetches. Once we reach A_{17}, we see an entry we have not yet prefetched, entering PRM again.

the indirect chain. Waiting mode prevents the core from re-entering piggyback runahead mode and avoids generating SVIs for overlapping iterations. A load can re-enter piggyback runahead mode when a load outside the prefetched range occurs. The following sections explain SVR's detailed working.

A. Microarchitecture

Figure 5 illustrates the SVR microarchitecture modifications over the baseline in-order core. To support SVR, we augment the baseline with the following structures (green boxes): (1) a stride detector for tracking loads with striding memory-access patterns, (2) a taint tracker for identifying the indirect chain from a striding load, (3) a register called *head striding load register (HSLR)* (in blue) to save the instruction pointer of the initiating (and terminating) striding load, (4) a last prefetch register (LP) to track the last address that SVR generated per striding load, to avoid overlapping prefetches, (5) a speculative register file (SRF) accessed by runahead instructions within the indirect chain, and finally, (6) the issue unit is modified to accommodate a scalar vector unit (SVU) which duplicates the current instruction and creates/issues up to N scalar copies of it. A loop bound detector (LBD) (in orange) and last compare register (LC) (in pink) are engaged when predicting the number of upcoming iterations that SVR must prefetch. The following sections explain the details of each structure.

1) Stride Detector: The stride detector uses a reference prediction table similar to that in stride prefetchers [17]; it is indexed with the PC of a load instruction. Figure 6 shows the fields of each entry of the stride detector. In SVR, the stride detector has multiple roles. Foremost, the stride detector is responsible for identifying load instructions with striding memory-access patterns. The stride is calculated as the delta between the 'Previous Address' and the current address, and is stored in the 'Stride' field. When the newly calculated stride is the same as the one already stored, the 'Saturating Counter' is increased to denote higher confidence. Second, the 'Last Prefetch' field in the stride detector prevents the core from entering piggyback runahead mode within the range fetched in the last round (waiting mode), to avoid duplicate address generation for redundant prefetches consuming compute resources. Third, the 'Seen' bit is used to detect inner loops in the presence of multiple (nested) striding loads by indicating if the another striding load has been seen before encountering

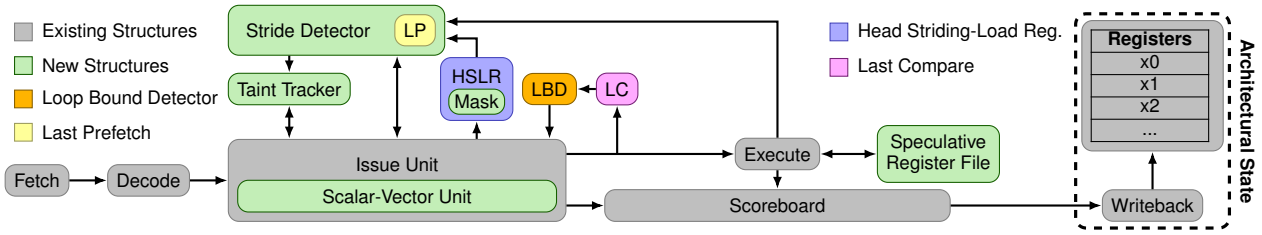


Fig. 5: SVR's microarchitecture: A conventional stall-on-use in-order processor is augmented with a stride detector to find candidates for scalar vector runahead, a taint vector to propagate vector inputs to generate new vector instructions, a slicer to generate multiple scalar-vector copies of the program's true instructions, and speculative registers to store intermediate results.

Fig. 6: Fields of a stride-detector entry: a reference-prediction table [17] to find future addresses to spawn vectors from, 'Last Prefetch' for waiting mode, a 'Seen' bit to track innermost loops, and fields to skip instructions past the final dependent load.

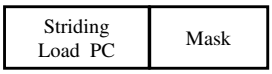


Fig. 7: Fields of the HSLR, storing the PC on which we start and terminate, and a control-flow mask for divergent lanes.

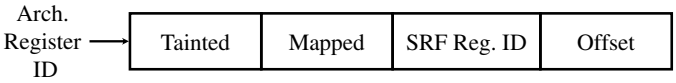


Fig. 8: Fields of a taint-tracker entry, stored per-architectural register. SVIs are generated if an input register is tainted (by another SVI writing to it, initially from a starting stride load), and is still mapped to an SRF ID (the SRF entry has not been replaced). Offset is used for LRU replacement.

the *head striding load* again. Finally, the stride detector also keeps track of the 'last indirect load (LIL)' in the chain.

Each load instruction accesses the stride detector when it reaches the head of the issue queue, and updates the detector upon execution. Upon reaching the head, a load instruction is issued as normal. However, if the load is detected as striding, the *HSLR* is updated with the load's PC, and the core enters into piggyback runahead mode. The issue unit issues N copies of the load, with addresses based on the original load and detected stride size. The following instruction in program order is issued only after all SVIs for a striding load have issued.

2) Detecting Indirect Loads: The indirect chain is identified using a *taint tracker*, which is indexed with architectural register identifier. Figure 8 shows the fields of an entry of the taint tracker. The 'Tainted' bit is set when an instruction is part of the indirect chain. The striding load sets the bit corresponding to its destination register. Any future instruction that reads a register with its tainted bit set in the taint tracker also sets the bit corresponding to its destination register. SVIs are generated for all the instructions that read a register with its tainted bit set, and the taint tracker is reset when the core leaves piggyback runahead mode. The 'Mapped' bit is set when the architectural register is mapped to a speculative register in the SRF, and the 'SRF Reg. ID' field stores the speculative register identifier. The 'Offset' field keeps track of the number of dynamic instructions between an instruction in the chain and the striding load; this field is used to implement LRU replacement on architectural-to-speculative register mappings if we run out of speculative registers (Section IV-A3).

3) Register Mapping and Recycling: SVR maintains a speculative register file (SRF) with 1,024-bit wide speculative registers, and 16 scalars can simultaneously access different 64-bit locations within each register. The mapping from architectural to speculative register is maintained in the taint tracker (Figure 8). There will typically be fewer SRF entries than architectural registers, as SRF entries are very wide, so are at a premium in terms of silicon area; this means there cannot be a one-to-one mapping between the two, and so we can only map a finite number of architectural registers at once. In piggyback runahead mode, architectural registers are allocated a free SRF entry on first write[1] (either via stride-load generation or because an input register is tainted), which taints and maps the architectural register in the taint tracker. The following scalars depending on the striding load read from this speculative register. When a mapped architectural register is overwritten by an instruction that is not part of the indirect chain, the 'Tainted' field is reset and the SRF register freed.

SRF entries are deliberately underprovisioned relative to architectural registers. To attempt to keep vectorizing if they are exhausted, we use LRU replacement; the SRF entry that is allocated to the least recently read (but still mapped) architectural register in the taint tracker is recycled. At this point, we indicate the invalidity of the old mapping by setting the architectural register's Mapped bit to 0 in the taint tracker, to prevent any instructions using the input register from being scalar-vectorized. The LRU replacement state is implemented via the Offset field; it is updated on each register read with the number of instructions that have executed since the start of the round of piggyback runahead mode.

4) Generating and Executing Scalar Vectors: SVIs are generated by the *Scalar Vector Unit* attached to the issue unit

[1]Unlike in an out-of-order core, which uses renaming to avoid name dependencies, only one copy of an individual architectural register can be live at once, so speculative registers are reused if a mapping already exists.

of the baseline core. Upon detecting a striding load, the Scalar Vector Unit creates an SVI: N further copies of the load with different addresses based on the memory address and stride size of the striding load, tainting the destination register in the taint tracker. It also does the same for any instruction with an input tainted in the taint tracker.

When we initiate piggyback runahead mode from a striding load (placed in the HSLR), a register keeps track of the 16 least significant bits of the *last indirect load* executed before we return to the HSLR load. When we terminate, this is copied into the HSLR PC's LIL entry in the stride detector[2]. We stop generating SVIs past the number of instructions in the LIL, or reaching the HSLR load, whichever comes first.

The copies are sent to the functional units alongside instructions from the main thread, and execute based on the availability of the relevant unit. The instruction from the main program has a priority over copies. We augment each entry of the scoreboard with a $\lceil \log_2(N+1) \rceil$ bit *return counter* to keep track of the N executing scalars. The return counter is set to N when issuing an instruction from an indirect chain, and a value of 0 in the return counter signals that all the scalars generated for the instruction have completed their execution. Upon return, each of the scalar instructions updates the SRF, and decrements the counter. Instructions dependent on the architectural or speculative destination can then proceed.

5) Termination and Restart: Piggyback runahead mode terminates when we reach the stride-load PC in the HSLR or a 256-instruction timeout. We then clear the taint tracker. At this point, the core enters *waiting mode* to avoid repeated work for the same striding load PC. A stride load cannot start a new round of runahead as long as the observed address is between the stride detector entry's *Previous Address* and *Last Prefetch* fields[3]. Once an address outside this range is observed at a given PC (due to discontinuity or reaching the address following *Last Prefetch*), we restart piggyback runahead mode.

6) Multiple Indirect Chains: In addition to the simple indirect chain in Figure 4, SVR can simultaneously handle multiple indirect chains. Figure 9 shows three cases with two indirect chains each, one starting from striding load A and another from striding load B. The three cases present further opportunity for improving performance by either generating prefetches for multiple chains or by retargeting to a chain with more timely prefetches (from a more inner loop). We use the 'Seen' field in the stride detector to detect other loops, which is set whenever the core detects a striding load. All the Seen bits except for the HSLR load are cleared whenever the core encounters the HSLR load, or retargets and resets the HSLR.

[2]With multiple concurrent strides (Section IV-A6), only the entry in the HSLR's stride detector is used and updated, but is shared between all concurrent stride chains, storing only the final dependent load in the set. In the presence of conditional code or aliasing, the LIL can change or be incorrect. 'LIL Confidence' (a 2-bit saturating counter) ensures that the core utilizes the field only for a constant last indirect load, by detecting through taint when we reach an alternative LIL once vectorization has finished.

[3]By storing a range that we performed runahead on, rather than having a countdown, we are able to detect when we overfetched on a previous loop, and so should restart a new round of runahead immediately because the values observed by the thread have not yet been prefetched.

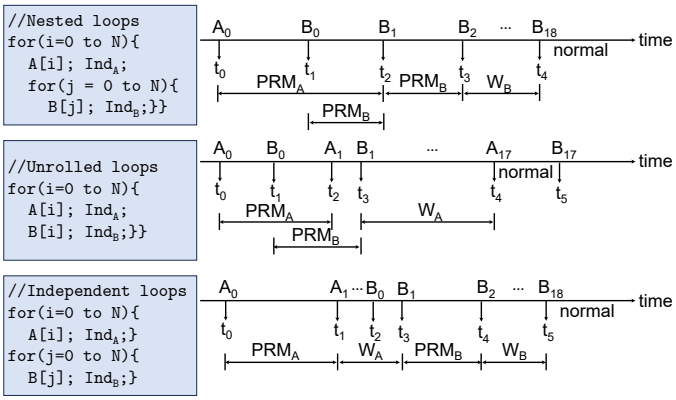


Fig. 9: Examples for multiple indirect chains handled by SVR. Accesses to arrays A and B are striding; PRM_i and W_i represent the time core is in piggyback runahead mode and waiting mode for load i, respectively.

Nested Loops. For nested loops, the core enters piggyback runahead mode for load A at t_0. The Seen bit in the load's stride detector entry is set, and HSLR is set to the PC of load A. When the core detects another striding load B at t_1, it sets Seen bit to 1 for load B, and begins piggyback runahead mode for load B. However, if the core detects another instance of load B before load A (at t_2), by detecting the Seen bit for load B already set, load B is part of a nested loop. Therefore, the core aborts piggyback runahead mode for both load A and load B at t_2, sets the HSLR to load B, and begins generating SVIs for the inner loop starting from load B.

Unrolled Loops. For unrolled loops with two indirect chains executing in parallel, generating prefetches for only one chain leaves performance on the table by missing the opportunity to prefetch memory accesses for another. Therefore, when generating prefetches for load A, if the issue unit finds another independent striding load B whose Seen bit is not set, it generates scalars for the indirect chain starting from B as well. The Seen bit for B is cleared when the core detects load A (the current load in HSLR) again at t_2. Therefore, unlike nested loops, we do not completely switch to generating scalars for B when we encounter another instance of B at t_3, simultaneously generating prefetches for both chains.

Independent Loops. For independent loops where we have completed piggyback runahead mode for HSLR load PC A, we leave A in the HSLR, and reset all 'Seen' bits every time we reach the HSLR regardless of whether the core is in piggyback runahead mode for load A. If we see a different load PC B for which the core is not in waiting mode we set its Seen bit, and if we subsequently see B with its Seen bit set, we retarget, update the HSLR with the PC of load B and begin piggyback runahead mode for load B, regardless of whether the core is in waiting mode or normal mode for load A. This bias towards the PC in the HSLR is to prevent retargets when first entering an inner loop from an outer loop from delaying runahead on the inner loop, by deprioritizing runahead on the outer loop, but still allowing retargeting to

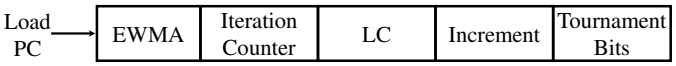


Fig. 10: Fields for loop-bound prediction, including loop-bound detector, EWMA and tournament.

any load we see twice, preventing starvation on new program phases with different loops.

7) Determining when SVR is Useful: To avoid SVR hurting performance more than it benefits, we monitor the accuracy of SVR using prefetch tags in the L1 cache, to track both the first use of a load brought in by SVR, and eviction before use. If, after a warmup of 100 uses or evictions, the accuracy drops below 50%, we prevent all loads from triggering SVR. To prevent this being permanent, we reset every one million instructions to give SVR another chance to execute.

B. Improving Efficiency

Here we improve the efficiency of SVR in two ways. First, we handle diverging vector lanes after a branch instruction. Second, generating a fixed number of scalars (for example, 16) can issue wrong prefetches if the real program has fewer upcoming iterations. Therefore, we use a predictor to decide the number of scalars that should be generated in each SVI.

1) Control Flow: In piggyback runahead mode, a striding load can feed a conditional branch instruction and, therefore, the generated scalars can follow different paths. While DVR [43] implements full reconvergence, since SVR piggybacks on existing instructions, it cannot execute down paths not covered by the real program. We instead mask diverging paths after the branch instruction, and generate SVIs only along the current path followed by the real instruction stream. During a round of piggyback runahead mode all vectorized instructions follow the same control path, regardless of which source stride load generated them. We have a single set of N mask bits (16 bits for 16-length SVR), associated with the Head Stride Load stored in the HSLR.

2) Loop Bound Prediction: Inaccurate out-of-bounds prefetches cause cache pollution and harm energy usage. Therefore, SVR predicts the upcoming number of iterations of a striding load with the help of a tournament predictor (Figure 10). The predictor decides the number of future iterations based on input from two techniques. First, a counter that keeps track of the *exponentially weighted moving average (EWMA)* of the number of iterations for each load instruction. Second, a *loop bound detector* that analyzes the past iterations of a loop and evaluates the upcoming iterations based on the register values used by the last compare/branch instruction that performs the bound check for the loop.

EWMA-Based Prediction. We keep track of the past weighted moving average of the number of contiguous iterations in the stride detector before a discontinuity, for each load instruction, and use that for deciding the number of scalars that must be generated. To catch the case where the number of contiguous strides is very large (so no throttling is needed), we also update the EWMA whenever we reach 512 entries or more, at which point the counter is reset. In the stride detector (Figure 6), every time the next seen address at a given PC matches `Previous Address + Stride`, an Iteration counter in the LBD is incremented. If a different address is seen, the Iteration counter is reset. On reset, `Iteration` is saved into the EWMA in the LBD, with the formula:

$EWMA_{new} = 7 \times EWMA_{old}/8 + Iteration/8.$

We fetch $min(EWMA - Iterations, N)$ elements each round if $EWMA - Iterations$ is positive, and $min(EWMA, N)$ otherwise, with N the SV length (16 default).

Loop Bound Detector. We also attempt to learn the loop bounds directly, so that we can tell the exact number of iterations the current loop will run for rather than relying on historical data. We do this by looking at the values input to backwards conditional-taken branches (conditional branches which the program too and whose target is before their source in the program, indicating a loop). Each Compare instruction updates the last compare register (LC) with its PC, source operand values (S.A and S.B) and source register IDs. If the flag destination is written to otherwise, the LC is reset. If we see a backwards conditional-taken branch to before the PC in the HSLR, with input from the LC's destination register, the loop bound detector (LBD) is trained. The LBD entry for the PC in the HSLR is looked up. If the Comp-PC in the LBD entry does not match the LC, then we decrement its confidence, and if it is zero we replace it with the LC entry, copying the source-operand values and register IDs from the LC into the LBD entry. If it does match, then we increment confidence and compare the old source-operand values with the new ones in the LC. If one of S.A and S.B changes but the other does not, then we treat the changing value as the loop increment (increase of induction variable each iteration) and the constant value as the loop bounds, and use the ratio between them to predict the total number of iterations[4].

On the first iteration of an inner loop from entry of an outer loop, typically both inputs will change, as the old entries will be stale, from the final iteration of a previous entry to the same loop. Rather than waiting to perform runahead until the next iteration, or proactively performing runahead at maximum length (which we show are not timely enough or accurate enough respectively), we attempt to proactively *scavenge* the loop bounds from the source registers later used by the compare instruction. When we reach the HSLR, and a stride-address discontinuity is observed, we read the registers that will soon be read by the compare instruction. The difference between the two Current Values (CV), divided by the loop

[4]This is similar to the loop-bound inference in DVR's [43] Discovery Mode. However, we can only passively observe in-order execution. On an out-of-order core, waiting a full iteration for the compare instruction to reach execute results in only a slight delay, whereas on an in-order core we must wait for the high-latency loads in the way (LBD+Wait in our evaluation), hence why we read registers themselves to boost performance. Likewise, without a nested discovery pass that can observe many loops at once, loop-bound-based throttling can be overly pessimistic if the next time we enter an inner loop it continues from the same place we left it. This is surprisingly common, happening in CC, PR and NAS-CG in our workloads, and the EWMA can observe and correct for it, by identifying that prefetches will still be accurate.

TABLE II: Hardware overhead of SVR, with the parameter N representing the length of the vector, and K representing the number of SVs. We assume SVR-16 with N=16 and K=8.

SD entry	48 bit PC 48 bit prev. address 8 bit stride dist. 2 bit stride conf.	48 bit LP 1 bit seen 16 bit LIL 2 bit LIL conf.
SD (32 entries)	$32 \times 173 = 5536$ bits	
TT entry (K=8)	1 bit tainted 1 bit mapped	$\lceil \log_2(K) \rceil = 3$ bits SRF Reg Id. 8 bit offset
TT (32 Arch. Reg)	$32 \times 13 = 416$ bits	
HSLR (N=16)	48 bit PC	$N = 16$ bit mask
SV (N=16)	$N \times 64 = 1024$ bits	
SRF (K=8)	$K \times 1024 = 8192$ bits	
LC	48 bit PC 64 bit val A 64 bit val B	 5 bit Reg. ID A 5 bit Reg. ID B
LBD entry	48 bit PC 9 bit EWMA 9 bit iteration counter	186 bit LC 16 bit loop increment 2 bit tournament
LBD (8 entries)	$8 \times 270 = 2160$ bit	
Scoreboard entry	$\lceil \log_2(N+1) \rceil = 5$ bit	
Scoreboard (add.) (32 entries)	$32 \times 5 = 160$ bit	
L1 Prefetch Tags	1024 bit	
Total	17738 bit = 2.17 KiB	

increment seen previously, is used as the prediction instead; the intuition is that the compare values are likely to have been initialized before the start of the loop, and are unlikely to be moved or spilled before they are used by the subsequent compare instruction. If these or any other assumptions are violated, we can fall back on the EWMA using a tournament.

Tournament Predictor. The tournament predictor uses 2-bit saturating counters to choose between EWMA and LBD. We continuously train both methods, but use whichever method the most significant bit indicates is best to issue piggyback runahead mode with. Right before the EWMA is updated (stride discontinuity or 512-element timeout), we check which predictor is closest to the true number of observed iterations, incrementing or decrementing the 2-bit counter as appropriate.

C. Hardware Overhead

Table II reports SVR's hardware overhead. With a default scalar-vector length $N = 16$, we see only 2 KiB of overhead. As N, the dominant impactor of MLP and thus performance, grows to 128, the SRF grows linearly to incur 9 KiB total overhead. We evaluate assuming $N = 16$ by default but give other values in our evaluation to quantify the performance-area trade-off.

V. Experimental Setup

We evaluate SVR using the Sniper v7.3 simulator [16]. The configurations can be found in Table III. The in-order core is based on the Arm Cortex-A510 [57], and the out-of-order core configuration is chosen to allow for the same number of in-flight instructions as the in-order core for fair comparison. There are no load and store queues in the in-order core as the instructions are issued in program order. Store-to-load dependencies are implicitly handled as a following (dependent)

TABLE III: In-order, SVR and out-of-order configurations.

Core	In-Order	SVR	Out-of-Order
Frequency Dispatch/commit width	2.0 GHz 3 Instr/cycle		
Scoreboard ROB Load/store queue Reservation station	32 — — —		— 32 16 32
Branch Predictor	hybrid local/global predictor, 10-cycle misprediction penalty		
Address translation	4 page table walkers 16-entry fully assoc. D-TLB 16-entry fully assoc. I-TLB 2048-entry 8-way S-TLB		
L1-I cache	64 KiB, 64 B cacheline, 4-way		
L1-D cache	64 KiB, 64 B cacheline, 4-way, 16 MSHRs, stride prefetcher		
L2 cache	512 KiB, 64 B cacheline, 8-way		
DRAM	50 GiB/s bandwidth, 45 ns latency		

load cannot bypass a prior (producer) store instruction. We also compare against IMP [60]. As mentioned in Section IV-C, the configuration for SVR uses 8 Speculative Registers, that are between 8 and 128 scalars wide (16 default), and 32 stride-detector entries. We use McPAT v1.0 [32] to evaluate power and energy consumption assuming a 22 nm technology node.

We evaluate SVR using three sets of workloads. The first set contains a variety of memory-latency-bound graph, database and HPC workloads: Camel [4], seq-CSR from the Graph500 benchmark suite [5], Hash-join [15] with a bucket size of 2 and 8, Conjugate Gradient and Integer Sort from the NAS parallel benchmarks [6], Kangaroo [4] as a derivative from NAS-IS, and finally randacc from HPC Challenge [33]. For each workload, we skip initialization and simulate 200 M instructions within the region of interest. The second set of workloads are from the GAP benchmark suite [11]: Betweenness Centrality (BC), Breadth First Search (BFS), Connected Components (CC), Page Rank (PR) and Single-Source Shortest Paths (SSSP). Five graph inputs were chosen for those benchmarks, two of which are synthetic, Kronecker (KR) and Uniform Random (UR), and the other three are real-world inputs from LiveJournal (LJN), Twitter (TW), and Orkut (ORK). The third set includes SPEC CPU2017 to evaluate SVR's overhead when there is no opportunity to vectorize.

VI. Evaluation

As previously reported in Figure 1, SVR achieves a 3.2× harmonic mean speedup relative to the baseline in-order core at its default 16-wide configuration, rising to 4.3× at 128-length. SVR comfortably beats the same system with an IMP prefetcher [60] (1.4×) since it can cover many more types of indirect memory access patterns. It also beats a full out-of-order core (1.3×). Moreover, since SVR is attached to a simple in-order core, achieves high performance (reducing static power consumption), and is accurate (unlike IMP), it is also by far more energy-efficient than competitor mechanisms.

A. Performance Breakdown

We report absolute cycles-per-instruction (CPI) — lower is better — in Figure 11 for each workload. This is as high as

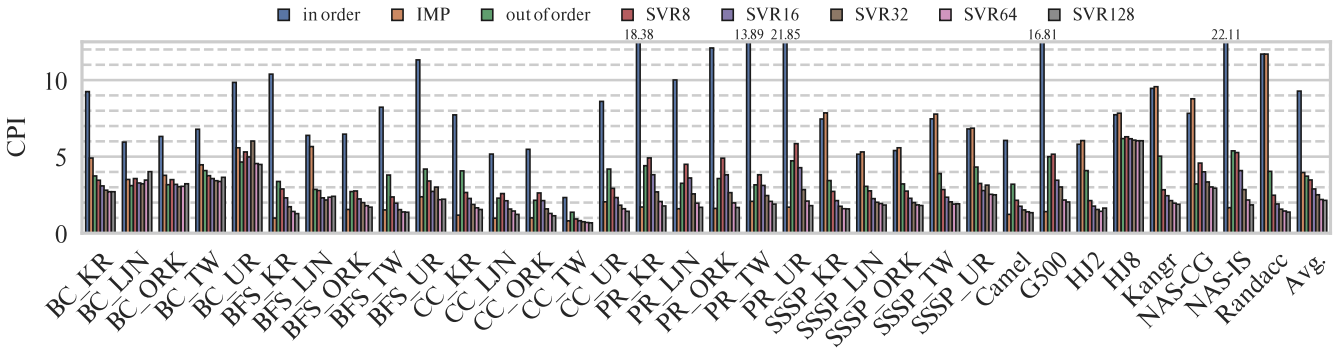


Fig. 11: Cycles-per-instruction (CPI) for in-order, IMP, out-of-order, and SVR with various widths (lower is better).

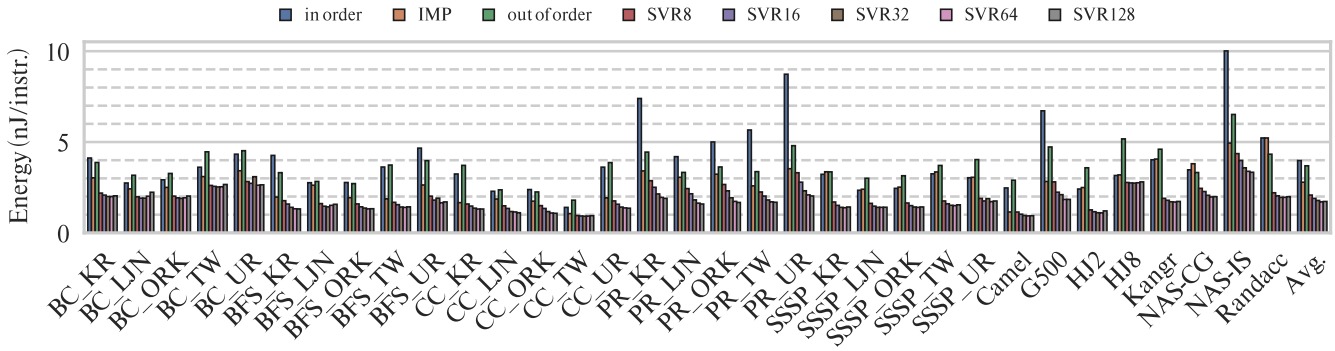


Fig. 12: Whole-system energy consumption per committed instruction (lower is better).

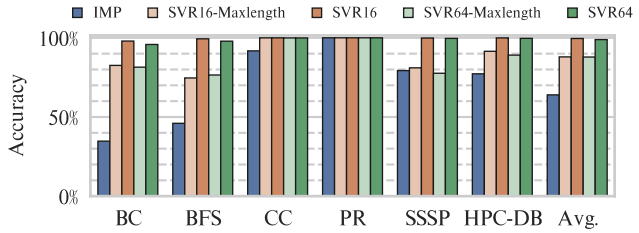
22.1 cycles per instruction for the in-order core, since it has almost no ability to overlap memory accesses. The out-of-order core avoids these spikes due to its reorder buffer allowing the issuing of multiple concurrent cache misses. However, even a 16-wide SVR outperforms it for almost every workload, by overlapping more memory accesses despite having none of the complexity of reorder-buffer or out-of-order issue logic. Longer SVRs are further able to saturate the memory system.

IMP fails to work for workloads without simple stride-indirect patterns (HJ2, HJ8, Kangaroo, Randacc, SSSP), so its performance is similar to the baseline in-order core. It can only partially handle the accesses in BC, and is inaccurate on BFS with the UR input, so it is slower than SVR despite gaining some benefit. However, for the simplest workloads that perform stride-indirect access patterns in long loops (BFS on Kronecker, Graph500, IS, and PR), IMP is able to outperform SVR. This is because, while SVR can overlap many memory accesses at once (particularly in its 128-wide configuration with higher 9 KiB area overhead), it cannot overlap memory accesses with computation, as it is still based on a stall-on-use in-order core, whereas IMP is a separate prefetcher that can bring in cache lines while the core computes. Still, even here SVR hides most of the CPI overhead of the in-order baseline.

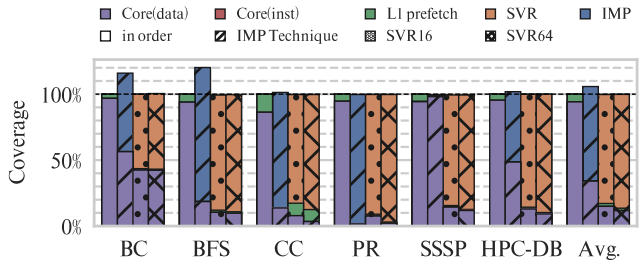
B. Energy

Figure 12 shows a breakdown per-workload of whole-system (system-on-chip and DRAM) energy consumption for each technique, per committed instruction. Despite the in-order core being significantly lower power than the out-of-order core (0.12 W and 1.01 W on average, respectively), the out-of-order execution is so much faster for these workloads that it is usually more energy-efficient under system-wide energy consumption due to the reduction in static power. Since IMP is on an in-order core, it typically incurs lower energy consumption than the out-of-order core despite being slightly slower, though this is less prominent on the workloads (BC, BFS) where it is inaccurate, particularly on the UR input which incurs short inner loops, due to the extra DRAM traffic. Even when IMP is accurate (PR, CC), it suffers slightly from redundant, overlapping L1-cache accesses from high-degree prefetches repeatedly requesting the next 16 elements each time a stride-load is seen, unlike SVR which avoids repeating requests, primarily to reduce compute demands, via waiting mode. SSSP is less efficient on both IMP and the out-of-order core than the baseline, since IMP is unable to capture its access pattern at all and the out-of-order core is not fast enough to recoup its power inefficiency.

By contrast, SVR is always the most energy-efficient technique by a considerable margin, since it is accurate, fast and simple. This is despite the fact that SVR effectively doubles the number of instructions executed while in piggyback runahead mode, by executing code once transiently and once for real. This is mitigated by its simple execution, where instructions are only fetched once, as well as by the fact that the core is memory-bound during these points, so the extra instruction count does not increase execution time or static power, and only accounts for 22% of the total power of the core.



a: Accuracy: proportion of prefetched cache lines accessed by the core within any cache before eviction from the LLC. SVR-Maxlength shows SVR without loop-bound prediction (Section IV-B2).



b: Coverage: proportion of loads that reach the DRAM controller from different origins, normalized to the in-order core; what exceeds 100% is caused by inaccurate prefetches.

Fig. 13: Accuracy and Coverage of SVR and IMP.

C. Prefetching Effectiveness

Accuracy. Prefetch accuracy is defined as the fraction of prefetched cachelines that are (later) accessed by the main program before being evicted from the LLC. Figure 13a shows that SVR's combination of loop bound detection and exponentially weighted moving averages allows it to be extremely accurate. Even without loop-bound prediction, SVR-16 still achieves 88% accuracy; the global accuracy threshold (Section IV-A7) turns off the prefetcher in workload phases where it struggles.

All techniques are accurate on PR, and all SVR techniques on CC. This is because their outer loops proceed in strict sequence over graph vertices, meaning the out-of-bound accesses for one inner loop are the in-bound accesses for the next inner loop. IMP, which has no access to the in-core information SVR uses to throttle prefetches, is otherwise consistently inaccurate, because it always fetches over the boundaries of inner loops by its maximum prefetch depth; by contrast, even unthrottled SVR overfetches less, as due to SVR's alignment properties, it will only go as far out-of-bounds as IMP if only one iteration of the loop is left when we start a new round of SVR.

Coverage. Figure 13b shows the coverage, or proportion of the baseline's memory accesses covered by SVR, along with the residual portion not hidden by SVR's prefetching, seen as demand misses by the CPU. In the simplest workloads (CC, PR), SVR's coverage is universal, as all memory access patterns get prefetched. In others (particularly BC and SSSP), there are cache misses in outer loops that SVR has no chance to vectorize, but SVR still captures the majority.

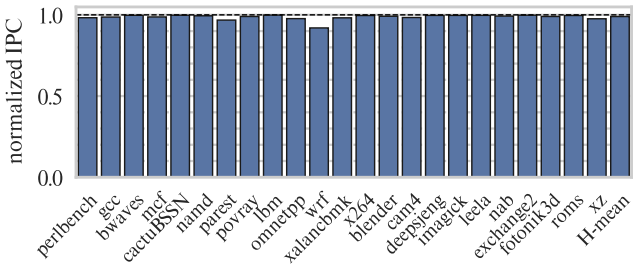


Fig. 14: SVR incurs a 1% average overhead for SPECrate 2017 when failing to find appropriate loops to vectorize.

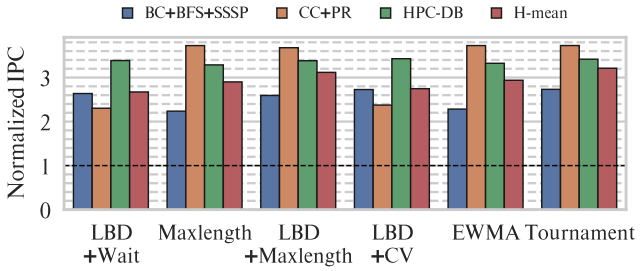
SVR-16 and SVR-64 have similar coverages; this is because they both cover the same loops. SVR-64's performance improvement comes from its improved timeliness, meaning more prefetches are overlapped, at the expense of slightly lower accuracy (Figure 13a) which results in a higher over-coverage (more memory accesses than the baseline). IMP's accuracy issues manifest in it going far outside the bounds of loop iterations, increasing DRAM accesses by up to 20%.

SPEC Benchmarks. While SVR is primarily targeted for workloads with indirect memory access patterns, it must not degrade the performance of other applications. We therefore evaluate SPEC CPU2017 benchmarks, see Figure 14. Overall, SVR's performance is close to the baseline in-order core (overall degradation of 1%). Only `wrf` experiences a performance degradation of more than 3%.

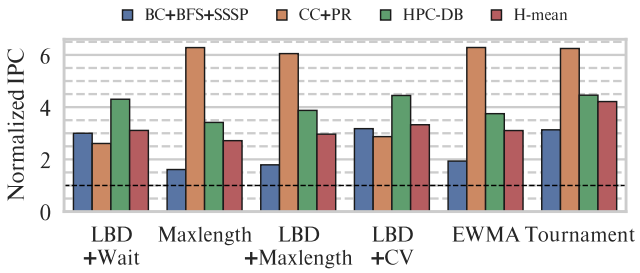
D. Comparison against Decoupled Vector Runahead

While SVR is closely inspired by DVR [43], DVR itself is infeasible to implement on an in-order core; as discussed in Section III, DVR requires a large physical register file and the ability for the core to execute multiple threads simultaneously, both of which would add considerable complexity. SVR is thus in part a direct reworking of DVR to allow it to work on an in-order core. However, we diverge in several key design decisions from what is optimal for DVR on large cores, and we consider the impacts of these decisions here.

Lockstep Coupling. SVR completely couples the runahead execution with normal execution. This hugely simplifies the design, by avoiding the need to support a second thread of execution, and by avoiding the need for a second scalar register file to store state. Compared to DVR, this loses us the ability to handle fully divergent control flow (Section IV-B1), supporting only masking instead. The cost of this is that HJ8 shows no speedup for SVR, and BC also sees a minor effect. However, it removes the need to copy the full register file before the start of a runahead, unlike in DVR. This is very significant in SVR for two reasons: (1) small cores have few write ports to copy registers, and (2) since SVR's widths are typically much smaller than DVR's, it happens unusually frequently. Modeling the cost of this register copy alone reduces SVR-16's performance improvement from 3.21× to 3.16×.



a: Normalized IPC for SVR-16



b: Normalized IPC for SVR-64

Fig. 15: Evaluating SVR's loop-bound prediction mechanisms.

Register Recycling. Since SVR has no spare physical registers, unlike DVR, we must store dedicated vector registers. Since each is a vector, we must store fewer than the pool of architectural registers to avoid high area overheads. This forces us to recycle registers (Section IV-A3) far more aggressively than DVR, which uses standard renaming techniques. SVR needs just two speculative registers to reach peak performance, whereas DVR's register-recycling policy needs eight. With DVR's policy and two speculative registers, SVR-16 drops from 3.2× to 1.9× speedup, and SVR-64 from 4.2× to 2.2×, due to heavy coverage loss in many workloads.

Loop-Bound Prediction Handling. Figure 15 reports performance (aggregated for similarly behaving workloads) for the various mechanisms inside SVR for loop-bound prediction (Section IV-B2, tournament default). DVR's Discovery Mode uses a policy similar to LBD+Wait (loop-bound detection, waiting until filled by the branch for the second loop iteration), to infer how wide we should vectorize. On an out-of-order core this happens almost immediately after triggering DVR, but on an in-order core it is delayed behind high-latency loads.

If we do no loop-bounds prediction (Maxlength) and always fetch 16 or 64 elements (for SVR-16 vs SVR-64), harmonic mean performance improves for SVR-16 but decreases for SVR-64, because in the latter case especially, the 50% accuracy threshold is hit, which blocks the prefetcher on many workloads.[5] Avoiding DVR's LBD+Wait's delay by using the LBD when ready but Maxlength otherwise improves performance slightly. However, because the inner loops are small for many workloads, accuracy banning is common enough for SVR-64 to suffer frequently, with the new scavenging of register Current Values before the first branch (LBD+CV) giving a large improvement.

[5]Likewise if the accuracy threshold is switched off, the inaccurate DRAM traffic causes untenably high overcoverage.

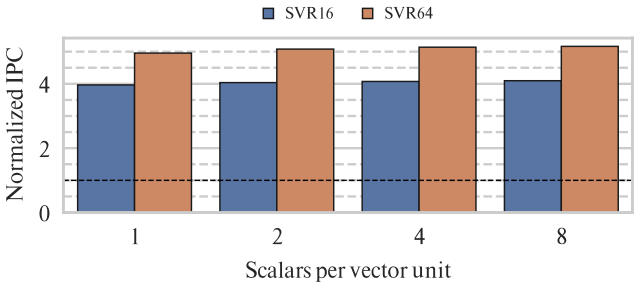


Fig. 16: Impact of increasing the compute vector width, or the number of elements within a scalar-vectors that go through execute simultaneously (default 1, unlike VR and DVR [40], [43] which are 8-wide). Despite SVR issuing one scalar at the time, instead of multiple, the performance is almost identical.

Still, for several workloads loop-bound detection alone is far too pessimistic, as it does not generate enough vector elements to maximize the width of the technique and thus overlap enough memory accesses to hide latency. DVR uses a two-dimensional *nesting* mechanism to collect input scalar registers from many outer loops at once. As well as being highly complex, this requires multiple scalar register files even if coupled, of which we only have one (not even enough for decoupling in one dimension). SVR's EWMA can detect this for the common case where each inner-loop iteration follows on directly from the last, and so even Maxlength is accurate; this is true for CC, PR and NAS-CG. For BC, BFS and SSSP, which benefited from decoupling but do not show this effect, we would expect to see a similar speedup were it possible to implement DVR's nesting on a simple core.

Finally, the Tournament achieves the best of both scenarios.

SVR with Vector Units. SVR executes using scalar, rather than vector, underlying instructions. This is so it can be used on the smallest cores, unlike VR and DVR [40], [43] which map execution onto 512-bit wide vector units such as AVX512 (which can pack 8 scalars into one executed instruction). Still, even in-order superscalars such as Arm's Cortex A55 have small Neon vector units of 128-bits [26], which would allow packing two instructions into one execution unit.

Figure 16 shows that being scalar instead of vector, in terms of execution, has no bearing on performance at all. Because the underlying core is in-order, and during piggyback runahead mode we deliberately saturate the memory system, execution during piggyback runahead mode is entirely memory bound. The compute throughput during this period thus does not change performance, so scalar execution is entirely sufficient.

Waiting Mode. Since SVR executes in lockstep, if it were eligible to re-enter SVR mode immediately after leaving it, like DVR is, then even for SVR-16, 15 lanes of 16 for every SVR execution would be repeats, causing an unfathomably high compute cost: SVR-16's speedup drops to 1.14× and SVR-64 drops to 0.56×, a slowdown. SVR adds waiting mode

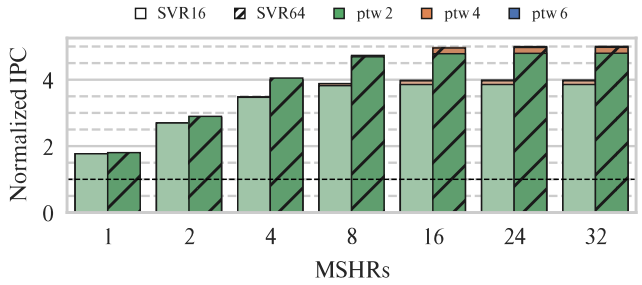


Fig. 17: Impact (harmonic mean speedup over in-order baseline) of varying the number of MSHRs and PTWs.

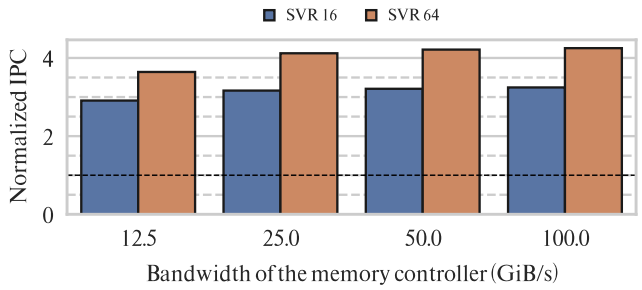


Fig. 18: Memory bandwidth sensitivity.

(Section IV-A5) to eliminate redundancy entirely, while still being able to restart SVR early if the program unexpectedly moves to an unfetched striding load.

E. Sensitivity Analyses

MSHRs and PTWs. Figure 17 shows how the speedup of SVR varies with an increasing number of miss-status handling registers (MSHRs) in the L1 cache and page-table walkers (PTWs). Even on the most constrained systems with one MSHR (so highly limited capability to perform hit-under-miss), SVR still speeds up the system, but is constrained by a lack of memory-level parallelism in the cache system. 16-wide SVR saturates at 8 MSHRs, whereas 64-wide SVR is able to overlap more memory accesses, saturating at 16 MSHRs. Only with very large numbers of MSHRs does page-table walking start to become the bottleneck, with a minor gain from 2 to 4 PTWs, due to the ability to handle increasing numbers of TLB misses simultaneously.

Memory Bandwidth. Figure 18 evaluates memory bandwidth sensitivity, relative to a baseline in-order core with the same bandwidth. SVR-64 benefits more from increased bandwidth than SVR-16 because it generates more memory requests during piggyback runahead mode. However, the overall performance improvement saturates with increasing bandwidth for both SVR-16 and SVR-64, which suggests that SVR does not saturate memory bandwidth. This is to be expected for a simple core coupled to high-bandwidth memory, and suggests that SVR across multiple cores simultaneously would give significant benefit.

VII. Related Work

A. Runahead Execution

Runahead execution [18], [36], [37] prefetches future memory accesses by pre-processing instructions transiently for distant loads. Naithani et al. [41], [42] improve performance by not flushing the ROB and only executing instructions used in future memory accesses. Hashemi et al. [23], [24], [25] perform a backward data-flow walk to extract the critical chain. Several techniques have improved the performance by deploying runahead with helper threads [50], using runahead to resolve future branches [49], and supporting runahead on multithreaded processors [56]. (Decoupled) Vector Runahead [40], [43] does not depend on linear speculation in the processor front-end to find future work. Instead, future loop iterations are independently vectorized to hide latency. SVR executes scalars on a simple in-order processor, and does not require a complex out-of-order frontend or backend to generate high levels of MLP.

B. Hardware Prefetching

Hardware prefetching is widely deployed to hide memory latency. Temporal prefetchers record the order of accesses to predict future loads, while typically incurring huge storage overhead [27], [29], [46], [58], [59]. Spatial prefetchers exploit the pattern of accesses to a memory region [7], [14], [17], [30], [51]. Many proposals improve the effectiveness (accuracy and coverage) of hardware prefetching [34], [47], [48].

A large body of work over the past decade has focused on prefetching indirect memory accesses (IMAs). IMP [60] was the first prefetcher at the L1 D-cache level to detect and generate prefetches for simple IMAs. DROPLET [10] coordinates between the last-level cache and memory controller. Pythia [13], Hermes [12] and SDC [28] propose skipping the cache, going straight to memory. Berti [44] prefetches IMAs and regular address patterns. Cache replacement policies have also been proposed to improve cache utilization for IMAs [8], [9], [19]. DMP [22] improves IMP's tracking on out-of-order processors by handling reordering, via differential matching between the indirect and striding access patterns. Unlike SVR which is in-core and is based on the real instruction stream, the success of these prefetchers inside the cache hierarchy is hampered by their limited ability to observe information.

C. Hardware-Software Coordinated Techniques

Prodigy [53] exploits the static program information from software and dynamic runtime information from hardware to prefetch indirect memory accesses. SWOOP [55] and Clairvoyance [54] exploit the decoupled nature of access and execute phases for improving energy efficiency on in-order cores. Walkers [31] reorder and overlap memory accesses for database workloads. Compiler-assisted prefetching techniques [1], [2], [3], [4] for indirect memory accesses either add extra instructions or extra cores to generate high MLP. SVR is purely microarchitectural, needing no compiler support.

VIII. Conclusion

Scalar Vector Runahead is an energy-efficient, high-performance CPU microarchitecture mechanism for graph, database and HPC analytics that extends simple in-order cores with minor alterations to allow them to achieve high memory-level parallelism through transient execution of independent operations. We hope that its 3.2× speedup will make efficient analytics at the edge practical. Further, we expect its energy efficiency and simplicity will drive the design of new forms of acceleration, by unshackling these workloads from the large out-of-order superscalars previously thought necessary to unlock their potential.

Acknowledgements

We thank the anonymous reviewers for their thoughtful comments and suggestions. This work is supported in part by Research Foundation Flanders (FWO) grant No. G018722N, European Research Council (ERC) Advanced Grant agreement No. 741097, and the Engineering and Physical Sciences Research Council (EPSRC) grant reference EP/W00576X/1. Additional data related to this publication is available on request from the lead author.

References

[1] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926254

[2] ——, "Software prefetching for indirect memory accesses," in *CGO 2017 - Proceedings of the 2017 International Symposium on Code Generation and Optimization*. Los Alamitos, CA, USA: IEEE Computer Society, feb 2017, pp. 305–317. [Online]. Available: https://doi.org/10.1109/CGO.2017.7863749

[3] ——, "An event-triggered programmable prefetcher for irregular workloads," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 578–592. [Online]. Available: https://doi.org/10.1145/3173162.3173189

[4] ——, "Software prefetching for indirect memory accesses: A microarchitectural perspective," *ACM Transactions on Computer Systems*, vol. 36, no. 3, jun 2019. [Online]. Available: https://doi.org/10.1145/3319393

[5] J. A. Ang, B. W. Barrett, K. B. Wheeler, and R. C. Murphy, "Introducing the graph 500." *Cray User's Group (CUG)*, vol. 19, pp. 45–74, 5 2010. [Online]. Available: https://www.osti.gov/biblio/1014641

[6] 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 *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 158–165. [Online]. Available: https://doi.org/10.1145/125826.125925

[7] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA '19. Los Alamitos, CA, USA: IEEE Computer Society, Feb 2019, pp. 399–411. [Online]. Available: https://doi.org/10.1109/HPCA.2019.00053

[8] V. Balaji, N. Crago, A. Jaleel, and B. Lucia, "P-OPT: Practical optimal cache replacement for graph analytics," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2021, pp. 668–681. [Online]. Available: https://doi.org/10.1109/HPCA51647.2021.00062

[9] V. Balaji and B. Lucia, "Improving locality of irregular updates with hardware assisted propagation blocking," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2022, pp. 543–557. [Online]. Available: https://doi.org/10.1109/HPCA53966.2022.00047

[10] A. Basak, S. Li, X. Hu, S. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, feb 2019, pp. 373–386. [Online]. Available: https://doi.org/10.1109/HPCA.2019.00051

[11] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: https://doi.org/10.48550/arXiv.1508.03619

[12] R. Bera, K. Kanellopoulos, S. Balachandran, D. Novo, A. Olgun, M. Sadrosadat, and O. Mutlu, "Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction," in *2022 55th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-55. Los Alamitos, CA, USA: IEEE Computer Society, oct 2022, pp. 1–18. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00015

[13] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1121–1137. [Online]. Available: https://doi.org/10.1145/3466752.3480114

[14] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "DSPatch: Dual spatial pattern prefetcher," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 531–544. [Online]. Available: https://doi.org/10.1145/3352460.3358325

[15] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core cpus," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 37–48. [Online]. Available: https://doi.org/10.1145/1989323.1989328

[16] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, aug 2014. [Online]. Available: https://doi.org/10.1145/2629677

[17] T. Chen and J. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 05, pp. 609–623, may 1995. [Online]. Available: https://doi.org/10.1109/12.381947

[18] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 68–75. [Online]. Available: https://doi.org/10.1145/263580.263597

[19] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, feb 2020, pp. 234–248. [Online]. Available: https://doi.org/10.1109/HPCA47549.2020.00028

[20] D. Folegnani and A. González, "Energy-effective issue logic," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 230–239. [Online]. Available: https://doi.org/10.1145/379240.379266

[21] A. Frumusanu. (2021) Arm announces mobile Armv9 CPU microarchitectures: Cortex-X2, Cortex-A710 & Cortex-A510. https://www.anandtech.com/show/16693/arm-announces-mobile-armv9-cpu-microarchitectures-cortexx2-cortexa710-cortexa510.

[22] G. Fu, T. Xia, Z. Luo, R. Chen, W. Zhao, and P. Ren, "Differential-matching prefetcher for indirect memory access," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2024, pp. 439–453. [Online]. Available: https://doi.org/10.1109/HPCA57654.2024.00040

[23] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Accelerating dependent cache misses with an enhanced memory controller," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 444–455. [Online]. Available: https://doi.org/10.1109/ISCA.2016.46

[24] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783764

[25] M. Hashemi and Y. N. Patt, "Filtered runahead execution with a runahead buffer," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 358–369. [Online]. Available: https://doi.org/10.1145/2830772.2830812

[26] M. Humrick. (2017) Exploring DynamIQ and Arm's new CPUs: Cortex-A75, Cortex-A55. https://www.anandtech.com/show/11441/dynamiq-and-arms-new-cpus-cortex-a75-a55/4.

[27] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 247–259. [Online]. Available: https://doi.org/10.1145/2540708.2540730

[28] A. V. Jamet, G. Vavouliotis, D. A. Jiménez, L. Alvarez, and M. Casas, "Practically tackling memory bottlenecks of graph-processing workloads," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024, pp. 1034–1045. [Online]. Available: https://doi.org/10.1109/IPDPS57955.2024.00096

[29] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 252–263. [Online]. Available: https://doi.org/10.1145/264107.264207

[30] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Los Alamitos, CA, USA: IEEE Computer Society, 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783763

[31] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the Walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013, p. 468–479. [Online]. Available: https://doi.org/10.1145/2540708.2540748

[32] 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 *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 469–480. [Online]. Available: https://doi.org/10.1145/1669112.1669172

[33] 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 *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 213–es. [Online]. Available: https://doi.org/10.1145/1188455.1188677

[34] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2016, pp. 469–480. [Online]. Available: https://doi.org/10.1109/HPCA.2016.7446087

[35] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 1–14. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00010

[36] O. Mutlu, H. Kim, Y. N. Patt, and J. Stark, "On reusing the results of pre-executed instructions in a runahead execution processor," *IEEE Computer Architecture Letters*, vol. 4, no. 01, p. 2, jan 2005. [Online]. Available: https://doi.org/10.1109/L-CA.2005.1

[37] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, ser. HPCA '03. Los Alamitos, CA, USA: IEEE Computer Society, feb 2003, p. 129. [Online]. Available: https://doi.org/10.1109/HPCA.2003.1183532

[38] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector runahead for indirect memory accesses," *IEEE Micro*, vol. 42, no. 04, pp. 116–123, jul 2022. [Online]. Available: https://doi.org/10.1109/MM.2022.3163132

[39] A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Decoupled vector runahead for prefetching nested memory-access chains," *IEEE Micro*, vol. 44, no. 04, pp. 20–26, jul 2024. [Online]. Available: https://doi.org/10.1109/MM.2024.3406891

[40] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector runahead," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. Los Alamitos, CA, USA: IEEE Computer Society, 2021, p. 195–208. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00024

[41] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout, "Precise runahead execution," *IEEE Comput. Archit. Lett.*, vol. 18, no. 1, p. 71–74, jan 2019. [Online]. Available: https://doi.org/10.1109/LCA.2019.2910518

[42] ——, "Precise runahead execution," in *Proceedings of the 26th International Symposium on High-Performance Computer Architecture*, ser. HPCA '20, 2020, pp. 397–410. [Online]. Available: https://doi.org/10.1109/HPCA47549.2020.00040

[43] A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Decoupled vector runahead," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 17–31. [Online]. Available: https://doi.org/10.1145/3613424.3614255

[44] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, "Berti: an accurate local-delta data prefetcher," in *2022 55th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-55. Los Alamitos, CA, USA: IEEE Computer Society, oct 2022, pp. 975–991. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00072

[45] S. Nayak, R. Patgiri, L. Waikhom, and A. Ahmed, "A review on edge analytics: Issues, challenges, opportunities, promises, future directions, and applications," *Digital Communications and Networks*, vol. 10, no. 3, pp. 783–804, 2024. [Online]. Available: https://doi.org/10.1016/j.dcan.2022.10.016

[46] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA '04. Los Alamitos, CA, USA: IEEE Computer Society, 2004, p. 96. [Online]. Available: https://doi.org/10.1109/HPCA.2004.10030

[47] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2020, pp. 118–131. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00021

[48] B. Panda, "CLIP: Load criticality based data prefetching for bandwidth-constrained many-core systems," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 714–727. [Online]. Available: https://doi.org/10.1145/3613424.3614245

[49] S. Pruett and Y. Patt, "Branch runahead: An alternative to branch prediction for impossible to predict branches," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 804–815. [Online]. Available: https://doi.org/10.1145/3466752.3480053

[50] T. Ramirez, A. Pajuelo, O. Santana, and M. Valero, "Runahead threads to improve SMT performance," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, 2008, pp. 149–158. [Online]. Available: https://doi.org/10.1109/HPCA.2008.4658635

[51] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 141–152. [Online]. Available: https://doi.org/10.1145/2830772.2830793

[52] R. Shioya, M. Goshima, and H. Ando, "A front-end execution architecture for high energy efficiency," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. USA: IEEE Computer Society, 2014, p. 419–431. [Online]. Available: https://doi.org/10.1109/MICRO.2014.35

[53] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. M. Morton, A. Ahmadi, T. Austin, M. O'Boyle, S. Mahlke, T. Mudge, and R. Dreslinski, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *2021 IEEE International Symposium on High-Performance Computer Architecture*, ser. HPCA '21, vol. 2021-February. Los Alamitos, CA, USA: IEEE Computer Society, feb 2021, pp. 654–667. [Online]. Available: https://doi.org/10.1109/HPCA51647.2021.00061

[54] K.-A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean, "Clairvoyance: look-ahead compile-time scheduling," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. IEEE Press, 2017, p. 171–184. [Online]. Available: https://doi.org/10.1109/CGO.2017.7863738

[55] K.-A. Tran, A. Jimborean, T. E. Carlson, K. Koukos, M. Själander, and S. Kaxiras, "SWOOP: software-hardware co-design for non-speculative, execute-ahead, in-order cores," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 328–343. [Online]. Available: https://doi.org/10.1145/3192366.3192393

[56] K. Van Craeynest, S. Eyerman, and L. Eeckhout, "Mlp-aware runahead threads in a simultaneous multithreading processor," in *High Performance Embedded Architectures and Compilers*, A. Seznec, J. Emer, M. O'Boyle, M. Martonosi, and T. Ungerer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 110–124. [Online]. Available: https://doi.org/10.1007/978-3-540-92990-1_10

[57] WikiChip, "Cortex-A510 - Microarchitectures - ARM." [Online]. Available: https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a510

[58] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 996–1008. [Online]. Available: https://doi.org/10.1145/3352460.3358300

[59] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 449–461. [Online]. Available: https://doi.org/10.1145/3307650.3322225

[60] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 178–190. [Online]. Available: https://doi.org/10.1145/2830772.2830807