Decoupled Vector Runahead

Ajeya Naithani
Ghent University
Belgium

Jaime Roelandts
Ghent University
Belgium

Sam Ainsworth
University of Edinburgh
United Kingdom

Timothy M. Jones
University of Cambridge
United Kingdom

Lieven Eeckhout
Ghent University
Belgium

ABSTRACT

We present Decoupled Vector Runahead (DVR), an in-core prefetching technique, executing separately to the main application thread, that exploits massive amounts of memory-level parallelism to improve the performance of applications featuring indirect memory accesses. DVR dynamically infers loop bounds at run-time, recognizing striding loads, and vectorizing subsequent instructions that are part of an indirect chain. It proactively issues memory accesses for the resulting loads far into the future, even when the out-of-order core has not yet stalled, bringing their data into the L1 cache, and thus providing timely prefetches for the main thread. DVR can adjust the degree of vectorization at run-time, vectorize the same chain of indirect memory accesses across multiple invocations of an inner loop, and efficiently handle branch divergence along the vectorized chain. DVR runs as an on-demand, speculative, in-order, lightweight hardware subthread alongside the main thread within the core and incurs a minimal hardware overhead of only 1139 bytes. Relative to a large superscalar 5-wide out-of-order baseline and Vector Runahead — a recent microarchitectural technique to accelerate indirect memory accesses on out-of-order processors — DVR delivers 2.4× and 2× higher performance, respectively, for a set of graph analytics, database, and HPC workloads.

CCS CONCEPTS

• **Computer systems organization → Superscalar architectures**; *Single instruction, multiple data.*

KEYWORDS

CPU microarchitecture, prefetching, runahead, speculative vectorization, graph processing

ACM Reference Format:
Ajeya Naithani, Jaime Roelandts, Sam Ainsworth, Timothy M. Jones, and Lieven Eeckhout. 2023. Decoupled Vector Runahead. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3613424.3614255

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614255

1 INTRODUCTION

Out-of-order cores are bigger than ever, with the latest processors featuring reorder buffers of many hundreds of entries [33]. And yet, although modern-day out-of-order (OoO) processors are given more than ample resources, and thus their out-of-order queueing resources are rarely filled to capacity, they are still memory-bound especially for workloads that feature chains of dependent memory accesses, or indirect memory accesses. One recent proposal, Vector Runahead [67, 68], presents a potential method for doing better. Rather than work-skipping as earlier runahead proposals do [29, 40, 62, 66] to keep uncovering memory-level parallelism, Vector Runahead reformulates the transient execution performed within runahead mode to be primarily based on loop-level parallelism, following independent groups of many different *dependent* chains of memory accesses from future loop iterations in the program, and running them in a vectorized manner to reduce front-end and back-end pipeline resource requirements.

Vector Runahead (VR) can successfully follow and prefetch the complex memory-access patterns in modern graph analytics, database and high-performance computing (HPC) workloads. However, like the underlying out-of-order core, even with a large reorder buffer (ROB), Vector Runahead is still memory-bound. Because the large reorder buffer rarely fills up, the resource starvation that triggers Vector Runahead rarely occurs, and so its benefits over even resource-bountiful out-of-order execution are not allowed to shine.

We propose *Decoupled Vector Runahead (DVR)*, which innovates over prior runahead proposals in several key ways. First, it completely decouples the runahead process from the main computation thread, by running it within a lightweight, in-order subthread context of its own, allowing initiation even when the core is not stalled on a full ROB, and allowing the main thread to continue to make progress on its intended computation. Second, building on VR, it implements GPU-style divergence and reconvergence on the many dynamically generated 'lanes' produced from the many future loop iterations within the speculative runahead context. Third, it performs a *discovery mode* within the main computation's thread to precisely predict how many loops into the future will be accessed, to limit inaccurate prefetches. When it has too few locations to prefetch from discovery mode alone, it performs *nested vector runahead* to generate inputs for many inner loop invocations from many different outer loop iterations simultaneously, which can then all be efficiently vectorized together to achieve extreme memory-level parallelism, even for workloads with complex data- and control-flow dependencies.

Decoupled Vector Runahead proactively prefetches cache-missing loads far in advance, meaning such loads do not sit in the reorder

```
for(i=0; i<NUM_KEYS; i++) {
    C[hash(B[hash(A[i])])]++;
}
```

Figure 1: Example indirect memory access pattern [67].

buffer stalling commit or preventing branches from being resolved, let alone stall the reorder buffer entirely. Performance improves substantially as a result of its accurate, timely prefetches. DVR means runahead is no longer an alternative to very large instruction windows for out-of-order processors [66]. In fact, it is much better by offering huge performance benefits even in addition to such a large instruction window. Our simulation results using a broad set of graph analytics, database, and HPC workloads report that DVR yields 2.4× and 2× higher performance on average (and up to 6.4× and 5.2×) compared to a baseline OoO core (with a 350-entry ROB) and Vector Runahead, respectively. We further demonstrate that the performance boost DVR offers is maintained when increasing ROB sizes, in contrast to Vector Runahead, thanks to its high accuracy, high coverage and timeliness, when prefetching many future dependent load chains in parallel and decoupled from the main thread.

2 BACKGROUND

2.1 Runahead Execution

Runahead execution [29, 40, 62, 66] prefetches future memory accesses into on-chip caches after the *instruction window* or *reorder buffer* of an out-of-order core fills up and stalls with a memory access at the head of the buffer. To avoid a long-latency memory access from stalling the core, it will evict the instruction from its reorder buffer, but continue with instructions after it. While these instructions will no longer be strictly correct, and will be rolled back later, the prefetches generated as a result are accurate, as it speculatively pre-executes the application's own future instruction stream. The processor stays in *runahead mode* for its *runahead interval*: the number of cycles from the full-ROB stall to the return of the long-latency memory access. Following this, it returns to normal (correct) execution mode.

Precise Runahead Execution (PRE) [69, 70] improves the performance of prior runahead techniques in three ways: (1) in runahead mode, it improves prefetch coverage by only executing chains of instructions that lead to full-ROB stalls, (2) it does not flush the reorder buffer when exiting runahead mode, therefore saving the penalty for flushing and refilling the pipeline, and (3) it can prefetch future memory accesses even for short runahead intervals. One key characteristic of all runahead techniques, including precise runahead, is that they depend on the processor front-end for delivering future instructions for the duration of a runahead interval. Consequently, the number of instructions executed in the runahead mode depends on the front-end width and runahead interval.

2.2 Indirect Memory Accesses

Many modern applications feature dependent memory accesses with complex address-calculation patterns and multiple levels of indirection. A simple example of such patterns is shown in Figure 1. Here, array A is accessed sequentially. However, the index to access array B is calculated by hashing the value at a particular index of A, and the index to array C is calculated by hashing the access to B. That is, accesses to C depend on accesses to B, which in turn depend on accesses to A. Accesses to B and C are termed the first and second levels of indirect memory accesses, respectively, and the chain of instructions between the access of array A and the access to array C is termed the *indirect chain*.

For workloads with indirect memory accesses, traditional runahead techniques fail to prefetch the majority of future memory accesses for two main reasons. First, even in the presence of a stride prefetcher, PRE cannot prefetch memory accesses beyond the first level of indirection [67]. For the example in Figure 1, depending on the work-skipping technique, the inputs to array C will either be invalidated [29], or fail to return before runahead terminates [69, 70]. Second, even for the first level of indirect memory accesses, the number of instructions (or the number of iterations of the loop) covered in runahead mode is limited by the width of the processor front-end and the runahead interval.

2.3 Vector Runahead

Vector Runahead (VR) [67] reinvents runahead execution — and alleviates the previously mentioned shortcomings — in three ways. First, it automatically generates instructions at different indices of an indirect chain, therefore eliminating the dependence of prior runahead techniques on the processor front-end for instruction supply in runahead mode. It then reorders those instructions such that many of them at a particular offset can be executed in parallel. This leads to all the load instructions at a particular offset being issued to the memory system simultaneously. Consequently, instead of waiting for one memory access to return — as typical runahead techniques like PRE do — the core waits for many memory accesses at the same time. Second, it groups a large number of reordered scalar instructions into vectors; this reduces the pressure on back-end resources, like the issue queue and execution units, to process instructions. Third, VR performs *delayed termination*, which only leaves runahead once memory accesses for an entire indirect chain have been generated, because it is faster at generating memory-level parallelism (MLP) than normal-mode execution.

In VR, the core enters runahead mode after a full-ROB stall. The process of reinterpreting scalars as vectors, or *speculative vectorization*, begins when the core encounters a striding load marking the beginning of an indirect chain. The processor vectorizes the stride load and its dependents to generate prefetches.

For the example in Figure 1, VR simultaneously generates accesses for multiple iterations of A (for example, from *i=0* to *63*) by reinterpreting the scalar load instruction accessing *A[0]* to a set of vector-gather instructions that access *A[0–63]*. Once this first set of loads returns, it begins the vectorization of the arithmetic instructions comprising the `hash()` function to calculate the indices *B[hash(A[0–63])]*. The gather instruction accessing *B[…]* accesses many different cachelines due to the indirect nature of accesses to array *B*, and therefore, instead of waiting for one memory access, the processor concurrently waits for 64 non-contiguous memory accesses. When these return, it generates memory accesses for all

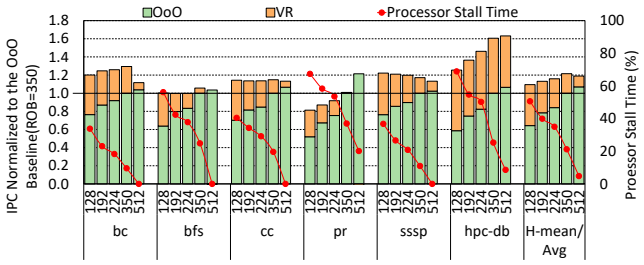

Figure 2: Performance of an OoO core and VR, normalized to a baseline 350-entry ROB OoO core (left axis), and processor stall time due to a full ROB (right axis), as a function of ROB size. *The performance gain of VR diminishes with increasing ROB size, and for some benchmarks overall performance even decreases.*

64 non-contiguous accesses to *C*. The processor then terminates runahead mode, as it has reached the last indirect load in the chain.

3 MOTIVATION

While Vector Runahead [67] is the first runahead technique to target indirect memory accesses and deliver substantially higher performance than prior runahead techniques, it is limited by the following factors:

(1) Performance Boost Diminishes with Bigger ROBs. Like all prior runahead techniques, VR waits for the reorder buffer to fill up. However, the size of the reorder buffer has consistently increased over recent years, and it therefore takes more cycles to fill up. As a result, the opportunity to enter runahead mode decreases with increasing ROB size, as reported in Figure 2. Indeed, processor stall time due to a full ROB reduces from 51% to 5% for an ROB size of 128 to 512 entries, respectively.

Reduced opportunity to enter runahead mode leads to a commensurate reduction in the performance boost VR offers. Figure 2 also reports performance for an OoO core and VR as a function of ROB size from 128 to 512 entries, normalized to our 350-entry ROB baseline OoO core (see Section 5 for the full experimental setup). While VR improves performance for all ROB sizes, and is faster than any out-of-order baseline no matter how small the ROB is, the performance benefit offered by VR diminishes with increasing ROB size. For some benchmarks this is so dramatic that *absolute performance actually decreases* with increasing ROB size. This is particularly the case for `sssp`, as well as `bc`, `bfs` and `cc` to a lesser extent. A smaller ROB triggers VR more often, which is faster than OoO execution and thus enables prefetching further down the future instruction stream. Decoupling from a full-ROB stall has the opportunity to trigger vector-runahead execution more frequently and hence deliver higher performance.

Key Insight #1: To maximize prefetching opportunity, VR must not wait for a full-ROB stall.

(2) Delayed Termination Stalls Commit. VR terminates runahead mode only after vectorizing the last load instruction in the indirect chain and generating prefetches for it. Meanwhile, it is likely that the load instruction that originally blocked the head of the ROB, and caused the ROB to fill up, has returned from memory. Although the OoO core can now commit instructions from the ROB, the processor does not return to normal mode, so as to allow the vectorized chain to complete first. This delayed termination stalls the commit stage on average 7.1% (and up to 11.8%) of the total execution time in VR across our set of benchmarks. This is a missed opportunity for the main pipeline to progress.

Key Insight #2: The process of vectorization and generating prefetches in runahead mode under VR must be decoupled from the main pipeline, so that the main core can also make forward progress while prefetching along the speculatively vectorized indirect chain.

(3) Cannot Adapt to Run-time Characteristics. Vector Runahead attempts to generate as many gathers for each scalar load as possible. The goal is to achieve high memory-level parallelism by keeping all the miss status holding registers (MSHR) occupied by the outstanding memory accesses. However, this assumes that the workload's induction-variable access, from which we spawn future dependent chains, continues to steadily increase far into the future. When we look at more complicated workloads, this assumption begins to falter, and yet they still exhibit memory-level parallelism.

Algorithm 1: Breadth-first search. *There are two strides (at lines 4 and 8) from which we can start Vector Runahead, resulting in a chain length of 4 or 2 respectively, and a highly data-dependent branch at line 9.*

```
1  Queue workList = {startNode}
2  Array visited[startNode] = true
3  while worklist ≠ ∅ do
4      Vertex V = workList.pop()
5      Edge E1 = Vertices[V]
6      Edge E2 = Vertices[V+1]
7      for Edge E=E1;E<E2;E++) do
8          Vertex W = edgeTo[E]
9          if !visited[W] then
10             workList.push(W)
11             visited[W] = true
```

Breadth-first search is a widely used graph-traversal algorithm that is used both in its own right and also as a kernel for finding connected components, maximum flows by the Edmonds-Karp algorithm [31], betweenness centrality [16], and many more. Algorithm 1 shows pseudocode matching the behavior of both the top-down step of GAP [12] and Graph500 [6]. In this workload, there are two possible points from which we can start Vector Runahead (two striding loads) at lines 4 and 8. Typically we will wish to vectorize from the latter, as it is an inner loop and so the accesses will be more timely. However, the length of this inner loop will be extremely data-dependent: not just on the size of the graph, but also its structure. Often, the loop will be far shorter than the amount we wish to vectorize by, and so Vector Runahead will fetch a significant amount of data the true execution will never access, polluting the cache and wasting DRAM bandwidth.

Key Insight #3: VR needs to (i) learn the data-dependent, dynamic number of iterations of each loop it runs, to avoid fetching useless

data, and (ii) update this each time it runs to respond to the latest run-time values.

(4) Inability to Vectorize Multiple Invocations of the Same Loop. If one iteration of a loop does not have enough loads to prefetch, the MLP exposed by VR is limited. It can begin the speculative vectorization of multiple invocations as they pass the main core but each time it only generates a small number of memory accesses, which are often generated by the core in the very near future anyway. To be able to fetch ahead far enough, VR must increase the degree of vectorization by discovering the correct values for multiple future versions of the same inner loop. In the breadth-first search example, this means we must be able to generate many stride accesses from line 4, and follow their dependencies through to line 8, in order to run not just many loads from within the loop, but many different versions of the inner loop from different outer loops simultaneously.
Key Insight #4: VR needs to look ahead to many future iterations of the same loop, if a single loop is dynamically determined to be too small to saturate the memory system, by skipping ahead to discover inputs to the same code from different outer loop iterations that will execute in the near future.

(5) Inability to Handle Control-Flow Divergence. Vector Runahead follows the control flow of the first scalar-equivalent instruction in the set of vectorized lanes, invalidating lanes with control-flow divergence. In the breadth-first search example, this is fine provided that the first edge in the sequence does not end in a previously visited vertex. Otherwise, we fail to execute prefetches for the operations inside the *if*-statement within the loop. In other workloads, such as betweenness centrality, there may be much broader divergence, with completely different memory accesses down each path.

Ideally, we should follow the true control flow of every single vector lane — and yet we still want to execute instructions as vectors whenever possible, getting the maximal use, and maximal parallelism, from each scalar-equivalent operation. To do this, we should take inspiration from GPUs, allowing threads to diverge and reconverge [54] when necessary.
Key Insight #5: VR should remove the constraint of control-flow matching between lanes, by supporting full SIMT GPU-style divergence and reconvergence.

4 DVR MICROARCHITECTURE

Decoupled Vector Runahead overcomes the shortcomings listed in the previous section as follows. When the core discovers that it is executing a loop with dependent loads, based on a striding load that can be used to predict future loop iterations, a specialized *vector-runahead subthread* is activated on the same core as the currently executing main thread. This subthread is dynamically generated to prefetch many memory accesses into the future, but without affecting the semantics of the main thread. The vector-runahead subthread runs alongside the main thread on the same core, much like how threads co-execute in simultaneous multithreading (SMT) [91], except that the subthread is microarchitecturally generated, transient (to prefetch into the cache rather than achieve real computation), speculative, reordered to achieve extremely high memory-level parallelism, and significantly simpler, i.e., the subthread executes in-order. The vector-runahead subthread is also closely related to simultaneous subordinate microthreading [20], which also aims at improving performance of the main thread. Whereas a subordinate microthread is written in microcode featuring specialized machine-specific instructions, the vector-runahead subthread is dynamically generated and derived from the main application thread.

To achieve high memory-level parallelism from this in-order vector-runahead subthread, even while following chains of dependent loads that stall the subthread, we use single-instruction multiple-thread (SIMT) data-level parallelism [54], to execute large numbers of each instruction from the front-end, each representing a different loop iteration, simultaneously, thereby prefetching far into the future. Since this happens continuously, and overlaps with the execution of the main thread, most of the main out-of-order thread's memory accesses hit in the L1 by the time it reaches them — thus even for very large processors with massive windows, significant speedups can be achieved.

Figure 3 provides a schematic of a processor's microarchitecture enhanced to support DVR. We explain the various components in the following sections.

4.1 Discovery Mode

To discover an induction-variable load that multiple future copies of a loop can be spawned from, as in the original Vector Runahead proposal [67], we use a stride detector to identify a striding load and its stride, i.e., a load that follows a regular address sequence. Once we have this information, we enter *Discovery Mode* to perform a series of new analyses. The purpose of Discovery Mode is to (i) check whether the striding load is the most suitable candidate for DVR, by being the innermost striding load, (ii) derive the loop bounds, to determine how many speculative vector prefetches to generate, and (iii) discover whether there are any dependent loads based on the striding load that can be suitably prefetched by the vector-runahead subthread. Discovery Mode follows the main thread's execution through one iteration of the loop, until it reaches the striding load again, at which point it exits Discovery Mode.

4.1.1 Innermost Striding-Load Detection. Once an initial striding load is detected and Discovery Mode is engaged, we follow the main thread's execution to detect other striding loads that could be better candidates for initiating vector runahead. In particular, we may discover a striding load that is part of a more inner loop, and thus whose future iterations will be more timely if we prefetch them during vector-runahead mode. Striding load detection is done using the *Reference Prediction Table (RPT)* [22, 63], which keeps track of all striding loads and their strides. We keep a register initialized to zero with one bit per RPT entry. Stride loads set their bit to 1. If already set, then we have seen the same stride-load PC twice during Discovery Mode before seeing the current target stride again. This means the new stride is more inner, so we switch to performing Discovery Mode on it instead, resetting this register, the VTT and FLR (Section 4.1.2). We can vectorize multiple strides in the same loop (e.g., caused by loop unrolling), and this process simply chooses one to be the trigger, preferring innermost strides.

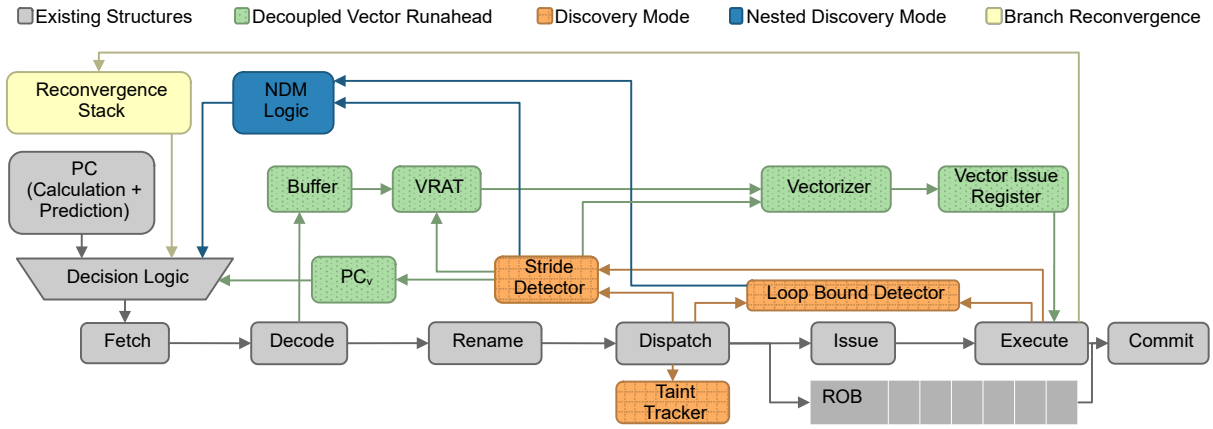

Figure 3: DVR processor pipeline. *The stride detector obtains information about loads from the dispatch and execute stages of the pipeline. Once a stride is detected, DVR enters Discovery Mode, which uses the Taint Tracker and Loop-Bound Detector to discover information for the subsequent runahead. The Nested Discovery Mode logic will be used if Discovery Mode finds too few elements of the loop to vectorize. Once Discovery Mode is complete, the vector program counter (PC_v) will be populated with the PC of the striding load, the VRAT will be populated with the striding load addresses and a copy of the main thread's scalar registers, and the decoupled vector-runahead subthread will initiate. The Reconvergence Stack will engage upon divergence in control-flow between the vector lanes.*

4.1.2 Dependent-Load Checking. For DVR to be worth triggering, it must bring useful data into the cache beyond that of a simple stride prefetcher [22], which we always assume such a system will have (and always leave enabled). This means there must be further loads dependent on the value identified via the stride detector for it to be worth initiating vector runahead. We use a small *Vector Taint Tracker (VTT)*, featuring a single bit per architectural integer register, to identify instructions that will later be vectorized. At the start of Discovery Mode, the VTT is initialized to all zeroes, except for the destination architecture register of the initiating striding load, which is set to one. This taint then propagates via instructions whose source register is tainted, transitively. If an instruction writes to a register whose taint bit is set but whose source registers are not, the taint bit of the target is reset. Whenever an input to a load is tainted in the VTT, the *Final-Load Register (FLR)* (initialized to zero at the start of Discovery Mode) is updated with the load PC. The FLR is a register that holds a single load PC, and its purpose is to identify the last load in the dependence chain originating from the striding load. The idea is then to vectorize all (tainted) instructions in the dependence chain starting from the striding load up until this last dependent load in the FLR. A non-zero FLR at the end of Discovery Mode indicates a load-dependence chain.

4.1.3 Loop-Bound Inference. The next step is to determine how many iterations are left for the inner loop to execute. This enables determining how many speculative vector prefetches to initiate during vector runahead. Doing so avoids generating wasteful and/or counterproductive loads that are out-of-bounds of the loop we expect to execute. During Discovery Mode, we look for the first branch with a backward edge, indicating a loop. The compare instruction that provides the source operand to this backward branch is used to determine the loop bound. In particular, we have both a Last-Compare Register (LCR) and a Seen-Branch Bit (SBB), which are zeroed whenever we update the Final-Load Register. If we see a compare instruction and the SBB is zero, we set the LCR with the compare's source and destination architectural register IDs. If we see a branch whose source matches the LCR destination and whose branch-taken destination is less than or equal to the striding load's PC,[1] then we set the SBB, to indicate that we should not alter the LCR unless we see a new final load.

We also take two checkpoints of the architectural register file: one upon entering Discovery Mode, and one upon leaving it. We then check the register mappings of the inputs to the identified compare instruction. If one stays constant for the whole Discovery Mode, and the other changes, we use (i) the constant value as the loop bound, and (ii) the difference in the changing value as the loop increment. This provides enough information to determine the remaining iterations of the loop. If we fail to produce a match, then we run for 128 elements, the limit for any invocation of DVR.[2]

4.2 Vector-Runahead Subthread Operation

Once Discovery Mode has identified a striding load, its stride, its dependence chain and the remaining iterations of the inner loop, the vector-runahead subthread is spawned once the main thread reaches the candidate striding load again. The subthread starts from the striding load and ends at the PC stored in the FLR, with the goal of speculatively prefetching a large number (up to 128 in our setup) of vectorized copies. In particular, the *Vectorizer* replaces the striding load by vectorized copies generated using its stride. Any instruction in the future instruction stream that depends on the striding load also gets vectorized.

[1] If we see other branches between the FLR and the LCR, we ignore the FLR and allow each runahead lane to continue onto the next stride PC, to allow it to fully explore any divergent paths that may manifest. The FLR is still used in Discovery Mode to help identify the loop, which must always encapsulate both the stride load and the FLR load.

[2] Runahead is transient execution and does not need to be correct, and so the goal for using more complex heuristics is only to reduce under/overfetching.

Vreg	Preg(s)							
R1	S45	S45	S45	S45	S45	S45	S45	S45
R2	V34	V35	V36	V37	V38	V39	V68	V69

Figure 4: An example VRAT allocation considering 8 physical registers (one per vector lane) for brevity rather than 16 as in our setup. *Architectural register R1 points to the same scalar physical register (S45) for all lanes. Architectural register R2 has been vectorized to 8 different vector physical registers, because either one of its sources was tainted, or control-flow divergence occurred.*

The subthread uses the same fetch, decode and execute units as the main thread. Subthread instructions are generated from the front-end buffer, which decouples the fetch stage from the rest of the pipeline by holding decoded micro-ops (eight in our setup). While subthread instructions use the same execution units, they use a different *Vector Issue Register (VIR)* — rather than an out-of-order instruction queue, as it is in-order — to handle execution of the vector instruction copies. An instruction in the vector-runahead subthread's issue register is issued whenever there is no instruction ready from the main thread for the same execution port.

4.2.1 Vector Register Allocation Table. The vector register allocation table (VRAT) stores the subthread's current mapping from architecture scalar registers to physical registers. Even though the subthread is in-order, we still need to rename its architectural registers because it shares the physical scalar and vector register files with the main thread. The VRAT stores multiple physical (scalar or vector) registers for each scalar architectural integer register. As illustrated in Figure 4, a scalar architectural register can be renamed to (i) the same scalar physical register in all vector lanes, in the case where the architectural register is not vectorized and there is no control-flow divergence across lanes, or (ii) multiple vector physical registers, where the architectural register has been vectorized or there is control-flow divergence.

To initialize the VRAT, all architectural registers from the main thread are allocated a fresh physical scalar register to decouple the subthread from its main thread. When the striding load is issued to the VIR, we allocate 16 vector (e.g., AVX-512) physical registers to map the load's target architectural register to. Unlike in an out-of-order processor, physical registers are not remapped with every new instruction, since the renaming is not trying to remove WAW nor WAR dependencies, i.e., the subthread executes in program order. Instead, we allocate new physical registers in only two cases. First, when one of the source registers has been vectorized (because it depends on the striding load), but the destination register has not yet been vectorized — at which point we must select 16 free vector physical registers to map to. Second, if the destination register is a vectorized register, but is about to be overwritten by a scalar instruction — this may occur as a result of a WAW dependence in the original program code — it is renamed to a scalar physical register from the free list. When only a subset of lanes are being executed, due to branch divergence, only some registers are renamed, as described in Section 4.2.3.

Physical registers are returned to the free list once they are overwritten. Overwritten registers are freed immediately, provided they are not used as a source register for the instruction to be

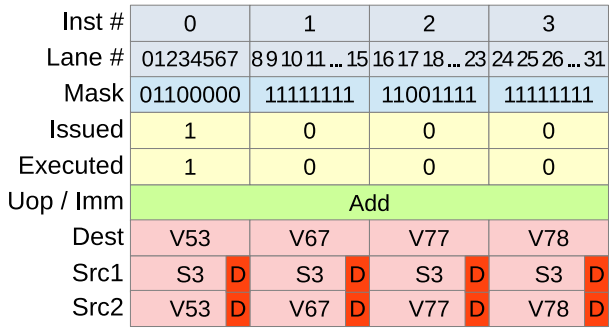

Figure 5: The Vector Issue Register showing 4 AVX-512 vector instructions (instead of 16 as in our setup for brevity). *Fine-grained masking has turned some scalar-equivalent lanes in AVX-512 instructions 0 and 2 into no-ops. The first AVX-512 instruction has been issued and executed, and the last three have neither been issued nor executed. Source register src1 is scalar register S3 and is shared among all lanes (none of which have diverged), which may be for example the base address of an array, whereas source register src2 has been vectorized (for example the index into the array). The destination registers are also vectorized, to the same location as src2 as they were the same architectural scalar register.*

issued — otherwise they are freed after execute, and tracked in the Vector Issue Register via the 'dead-source' bits (since it occurs after the overwriting occurs within the VRAT), as discussed in the next section.

4.2.2 Vector Issue Register. To achieve a significantly higher degree of memory-level parallelism than a single vector register (8 64-bit loads, as for AVX-512), we overlap the execution of multiple vector copies of the same instruction, with the target of achieving 16 AVX-512 vectors (or $16 \times 8 = 128$ scalar-equivalent loops) in-flight simultaneously. Instead of using a scalar issue queue, we use a single Vector Issue Register (VIR), responsible for the issuing of each vector copy of the scalar instruction (Figure 5).

If all inputs to the instruction are scalars, then just a single scalar instruction is issued. If the instruction is marked as a striding load, we use the stride detector to fill in all 128 values, and issue these as 16 vectorized AVX-512 loads. If the instruction depends on at least one vectorized input, we likewise issue 16 vectorized copies of the instruction in sequence to the execution units. Vectorized instruction copies are issued to the execution units whenever a suitable unit is free (not being used by the main thread). Within one AVX-512 instruction, we have 8 mask bits, to indicate lanes where one of the sources has been marked invalid, either through a fault, through use of floating-point registers, or through control-flow divergence. Some lanes may start as masked out, if Discovery Mode's loop-bound inference predicts that there will be less than 128 scalar-equivalent loops it can fetch. Once all instruction copies have issued and executed, if the 'dead-source' bit is set on any of the sources, the physical registers are freed. Then, we fetch the next instruction, and repeat.

Vectorized load instructions are treated like vector gather operations [87]: they are split into scalar loads in the LSQ and sent to the

PC (48 bits)	Mask (128 bits)
0x1234	111111100000
0x12a0	000000011111

Figure 6: An example reconvergence stack. *The top of the stack stores the current PC and mask. Once the reconvergence point is reached, the stack head is popped and execution proceeds with the next PC and mask.*

cache hierarchy individually. The memory system handles them concurrently with other regular scalar loads, allocating a different MSHR.

4.2.3 Branch Reconvergence. Dependent loads may be conditional, i.e., they appear down some control-flow paths and not others inside the inner loop. We allow each scalar-equivalent lane to diverge from the others. We therefore use a GPU-like reconvergence stack [54]. The results of the branches in all active lanes are compared against each other. If the next PC for any lane diverges from the others, we split the lanes based on their new destination, generate masks based on common groups, and place the masks and target PCs onto a reconvergence stack (Figure 6). We follow the first lane all the way to the reconvergence point, which we set to the vector-runahead termination point (Section 4.2.4), to avoid special tracking. Once we reach the termination point for a set of matching lanes, we pop the head off the reconvergence stack, reset the masks, and proceed from the next PC in the stack.

Each lane is simultaneously mapped in the VRAT. If we have divergence in scalar renaming (because we use different scalars), and this divergence occurs neatly across AVX-512 instruction boundaries, then we overwrite each scalar according to which of the 16 AVX-512 instructions use it. If we have divergence in scalar renaming within an AVX-512 instruction, we convert the destination to an AVX-512 physical register, and copy the scalar values being replaced.

4.2.4 Termination. The vector-runahead subthread terminates when the lanes reach the final indirect load in the sequence (identified by the FLR), or the next iteration of the stride PC in the case of divergence, with a 200-instruction timeout (in case we leave the loop entirely in a way not picked up by the loop bound detector, e.g., via a `break`).

The main thread executes concurrently with the vector-runahead subthread. Once the subthread has terminated, the main thread again becomes eligible for entering Discovery Mode the next time it executes a striding load, and thus for re-initiating DVR. The main thread will have made significant progress by this point, and most of its cache accesses will become L1 hits, provided the DVR subthread was accurate and timely.

4.3 Nested Vector Runahead

Loop-bound inference (Section 4.1.3) provides an accurate count of how many iterations each loop will execute, and thus how many scalar-equivalent lanes DVR can fill with useful prefetches. This may well be significantly lower than the 128-element maximum we can achieve, if each inner loop is relatively short, hurting the total memory-level parallelism, and thus limiting the benefits of the latency overlapping achieved by DVR.

The goal of the Nested Vector Runahead is to find iterations from multiple *invocations* of a loop when the loop bound detector does not find enough upcoming iterations of the innermost striding load (Section 4.1.1). The Nested Vector Runahead benefits benchmarks with patterns shown in Algorithm 1. If the `for` loop at `line 7` has a small number of iterations, vectorizing the chain starting from the *inner* striding load at `line 8` cannot generate high MLP. Therefore, it is critical to prefetch indirect chains from many invocations of the `for` loop. Nested Vector Runahead works in two steps. First, it performs a *Nested Discovery Mode (NDM)* to vectorize the chain of instructions from the *outer* striding load to the *inner* striding load, and discover loop bounds and data inputs to multiple invocations of the inner loop. Second, upon reaching the inner striding loop, it expands vectorization further to cover the inner loop as well.

4.3.1 Nested Discovery Mode. The goal of the NDM is to find the starting striding addresses and loop bounds for many different invocations of the inner loop at the same time. During a discovery mode (Section 4.1), the loop-bound detector may find fewer than 64 upcoming iterations of a loop. In this case, once the vector-runahead subthread is spawned, instead of performing vector runahead immediately, we alter the direction of the branch with the backward edge (see Section 4.1.3) and begin NDM on the in-order subthread by setting `PCv` to the instruction following the branch (not-taken path instruction). The subthread runs concurrently with the main thread. We still save both the source registers in the LCR. The constant loop increment and address of the striding load are saved in two new registers called *Increment Register (IR)* and *Inner Load Register (ILR)*, respectively.

The NDM subthread begins executing scalar operations, but skips all the upcoming iterations of the inner loop due to the altered branch direction, and executes instructions outside the inner loop. When it finds an outer striding load with an address smaller than the address in the ILR (e.g., line 4 versus line 8 in Algorithm 1), it performs its first vectorization step: it vectorizes the striding load (by a factor of 16, to attempt to find at least 128 viable inner loop iterations) and marks the load's destination in the taint vector.

The process of vectorization continues for the dependents of each outer striding load — until it reaches the first iteration of each inner striding load. In Algorithm 1, the outer striding load at `line 4` has dependents at both `line 5` and `line 6`.

When it it reaches the inner striding load (at `line 8`), it reads the values of the vectorized copies of the source registers in the LCR, and uses these and the value in IR to calculate the number of invocations of the inner loops for each of our vectorized outer loops. If no outer striding load with an address lower than the inner striding load appears within 200 instructions after entry to the NDM, the subthread re-calculates the loop bound based on the values in LCR and IR, and vectorizes the inner striding load by the loop bound. That is, the subthread resorts back to the number of iterations calculated by the loop bound detector during the initial discovery mode.

4.3.2 Further Vectorization. Based on the loop bounds detected, the NDM subthread then collects as many striding inner addresses as possible with a maximum limit of 128. Addresses beyond the first

Table 1: Baseline configuration for the OoO core.

Core	4.0 GHz, out-of-order
ROB size	350
Queue sizes	issue (128), load (128), store (72)
Processor width	5-wide fetch/dispatch/rename/commit
Pipeline depth	15 front-end stages
Branch predictor	8 KB TAGE-SC-L
Functional units	4 int add (1 cycle), 1 int mult (3 cycles), 1 int div (18 cycles), 1 fp add (3 cycles), 1 fp mult (5 cycles), 1 fp div (6 cycles)
Vector units	3 ALU, 2 shift, 2 add, 2 mul, 2 shuffle
Register file	256 int (64 bit) 256 fp (128 bit) 128 vector (512 bit)
L1 I-cache	32 KB, assoc 4, 2-cycle access
L1 D-cache	32 KB, assoc 8, 4-cycle access, 24 MSHRs, stride prefetcher (16 streams)
Private L2 cache	256 KB, assoc 8, 8-cycle access
Shared L3 cache	8 MB, assoc 16, 30-cycle access
Memory	50 ns min. latency, 51.2 GB/s bandwidth, request-based contention model

128 are discarded. The NDM subthread then performs vectorization from the inner striding load, by populating its vector registers with these 128 targets, with all other registers set based on which of the 16 outer-loop lanes it was spawned from (scalar for currently untainted registers, and vector for registers tainted in NDM). It taints the destination of the inner striding load and enters DVR with each lane, starting and terminating as specified in Section 4.2.

4.4 Hardware Overhead

The hardware structures to support DVR incur only 1139 bytes overhead. The 32-entry stride detector requires 460 bytes: each entry incurs 48 bits for the load PC, 48 bits for the previous memory address, 16 bits for the stride distance, 2 bits for the saturating counter, and 1 bit for innermost detection. The VRAT is a 16-entry table (288 bytes): each entry features 16 register identifiers each requiring 9 bits (to select one of the 128 vector physical registers and 256 integer physical registers). The VIR incurs 86 bytes: 128 bits for the mask, 16 bits issued, 16 bits executed, 64 bits uop and imm, 9×16 bits for the destination, 10×16 bits for src1, 10×16 bits for src2. The front-end buffer incurs 64 bytes for 8 micro-ops. The 8-entry reconvergence stack requires 176 bytes: 6 bytes for the PC and 128-bit mask for each PC. The FLR and LCR require only 6 bytes and 2 bytes, respectively; the SBB requires only 1 bit. The loop-bound detector saves two checkpoints (2×16×8 bits for the register ID mappings) and two registers for the compare and branch instructions, totalling 48 bytes. The taint-tracker needs 16 bits. For NDM, the IR and ILR require 7 bits and 6 bytes for keeping track of the loop increment (maximum 128) and ID of the address of the inner striding load.

5 EXPERIMENTAL SETUP

Simulation Infrastructure. We use Sniper 6.0 [18], an x86 simulator with its most detailed, cycle-level core model to simulate an aggressive 5-wide 350-entry ROB superscalar, out-of-order processor. The configuration of the core, the key microarchitectural structures of which are inspired by Intel Ice Lake processors [36], is provided in Table 1 [28]. A hardware stride prefetcher is always enabled at the L1-D cache level. Additionally, there are 24 MSHRs to keep track of outstanding misses from L1-D. The branch predictor is the 8 KB TAGE-SC-L from the 2016 CBP [83].

Table 2: Graph inputs used for the GAP suite [12].

Input	# Nodes (in Millions)	# Edges (in Millions)	LLC MPKI
Kron (KR)	134.2	2111.6	19
LiveJournal (LJN)	4.8	69.0	21
Orkut (ORK)	3.1	1930.3	18
Twitter (TW)	61.6	1468.4	61
Urand (UR)	134.2	2147.4	32

Benchmarks. We evaluate a total of 13 benchmarks from the graph analytics, database, and HPC domains featuring complex address-calculation patterns for a chain of indirect memory accesses. Five of the benchmarks are taken from the GAP benchmark suite [12]: Betweenness Centrality (`bc`), Breadth-First Search (`bfs`), Connected Components (`cc`), PageRank (`pr`), and Single-Source Shortest Path (`sssp`). Eight benchmarks, namely `Camel`, `Graph500`, Hashjoin with two and eight hashes (`HJ2` and `HJ8`), `Kangaroo`, `NAS-CG`, `NAS-IS`, and `RandomAccess`, are primarily from the database and HPC domains; these benchmarks have been extensively used by prior work [2, 3, 67, 88, 89], and we collectively call them `hpc-db` (high-performance computing and databases benchmark). Table 2 describes graph inputs; LLC MPKI shows the number of misses per kilo instructions aggregated over the five benchmarks for each input on our baseline OoO core. We use the region-of-interest (ROI) marker utility in Sniper to skip the initialization phase for each benchmark and simulate the next representative 500 M instructions.

6 EVALUATION

We evaluate the following runahead techniques relative to our baseline OoO core:

- Precise Runahead Execution (PRE) [69]: The state-of-the-art runahead technique that selectively executes only the chain of instructions leading to long-latency loads, and recycles register-file and issue-queue resources dynamically to avoid pipeline flushes.
- Indirect Memory Prefetcher (IMP) [98]: The indirect memory prefetcher, that works at L1 D-cache level and prefetches indirect memory accesses originating from striding access patterns.
- Vector Runahead (VR): The first vector-runahead mechanism proposed by Naithani et al. [67].
- Decoupled Vector Runahead (DVR).
- Oracle: A hypothetical technique that knows all memory accesses in advance, and prefetches them at the appropriate point in time to avoid stalling.

6.1 Performance

Figure 7 reports normalized performance for each technique on every benchmark-input combination. PRE rarely yields more than

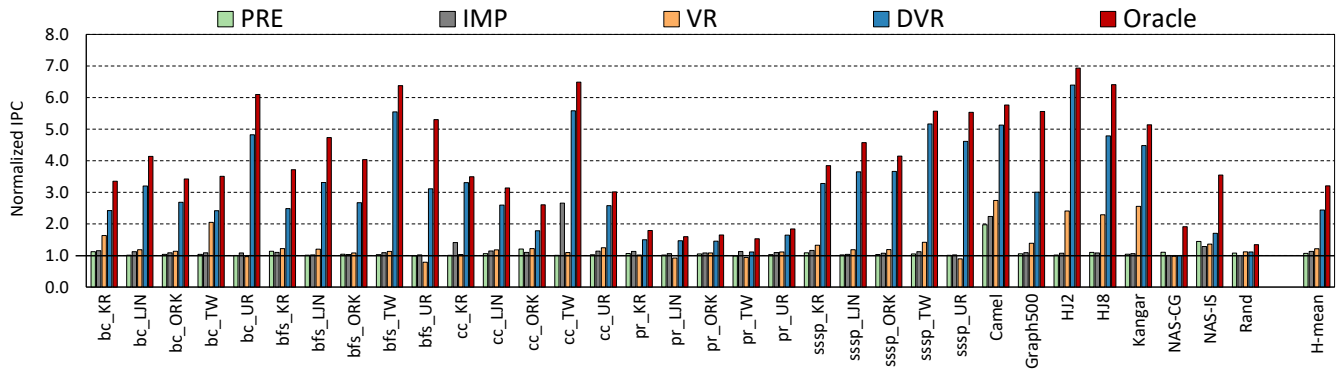

Figure 7: Performance for PRE, VR, DVR and Oracle normalized to a baseline OoO core. *DVR achieves* 2.4× *higher performance (and up to* 6.4×*) compared to a baseline OoO core.*

negligible performance improvements (with `Camel` and `NAS-IS` as exceptions). IMP performs better than PRE as it can detect simple-indirect patterns in benchmarks such as `cc`, `Camel`, and `NAS-IS`. However, it cannot prefetch indirect accesses for other benchmarks with more complex address calculation patterns. Vector Runahead manages slightly more (1.2× harmonic mean) because it is able to follow, reorder and vectorize the chains. Still, both PRE and VR suffer on large cores. The ROB rarely fills up, and even though there is still potential performance to be gained, the fact that neither PRE nor VR often reach their trigger condition limits their speedup. This is especially pronounced on the GAP benchmarks, where frequent branch mispredictions imply that the reorder buffer rarely reaches full utilization before the misprediction is discovered. This is the reason why IMP, which is detached from the core size and works at L1 D-cache level, performs better than VR for benchmarks such as `cc_KR` and `cc_TW`. In some cases where Vector Runahead is triggered, it decreases performance because of its inaccuracy (e.g., `bfs` on the `UR` dataset): when inner loops are short, the lack of DVR's Discovery Mode evicts useful data from the cache and wastes DRAM bandwidth. DVR often yields close to Oracle-level performance; it is more proactive in generating prefetches than VR and PRE, and achieves a 2.4× average speedup and 6.4× maximum.

Granted, there are still some workloads where DVR does not reach the full potential of a perfect Oracle, since it is not given full knowledge of the future or unlimited resources. In some cases (`NAS-CG` and `NAS-IS`), the workload is so simple that looking ahead only 128 elements into the future is insufficient to hide the full memory latency on such a large core: wider 256-element DVR units would achieve the higher performance of the Oracle, at the expense of a larger VRAT and more physical vector registers being required to be mapped simultaneously. In others, the memory-level parallelism is more difficult to find. This is particularly pronounced on workloads running the `UR` graph, where vertices are uniformly smaller than the 128-edge-element target, used by DVR within inner loops to generate MLP, unlike the power-law graphs (KR and Graph 500) which spend more time in highly populated vertices. As we shall see, Nested Vector Runahead mitigates this issue partially but still suffers from timeliness due to the complex dependencies.

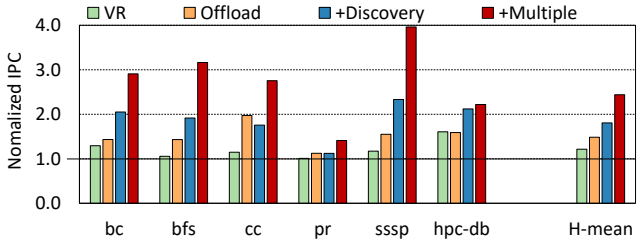

Figure 8: Breaking down DVR's performance normalized to the baseline OoO: (1) Vector Runahead [67], (2) Offload triggers a vector-runahead subthread whenever a stride is detected, (3) Discovery Mode further improves prefetch accuracy, and (4) Nested Runahead Mode completes DVR by further increasing memory-level parallelism over short loops.

6.2 Performance Breakdown

Figure 8 shows how the constituent parts of DVR contribute to the overall performance gain. Offloading Vector Runahead to a subthread, and thus allowing it to run more proactively than just on a full ROB, gives large benefits on its own: from 1.2× with a base Vector Runahead to almost 1.5× here. Indeed, the fact that the base Vector Runahead is out-of-order and the offloaded DVR is in-order is barely relevant when it comes to performance: each scalar-equivalent instruction in DVR does so much work, and brings in so many (vectorized gather) loads that there is no need for full out-of-order execution in the vector-runahead subthread.

Adding Discovery Mode particularly benefits `bc`, `bfs` and `sssp`; the over-fetching that vector-runahead techniques otherwise cause results in enough cache pollution and bandwidth wastage for the more accurate Discovery Mode to win out. Still, it is a double-edged sword on `cc` and `pr`, where the wrong-path execution triggered by DVR without Discovery Mode happens to bring in the correct data despite being out-of-bounds, as each outer loop generates only sequential values for the inner loop, unlike `bc`, `bfs` and `sssp`. Still, the full DVR technique, completed with the addition of Nested Runahead Mode, is uniformly best, because it can most effectively generate MLP far into the future even for short inner loops.

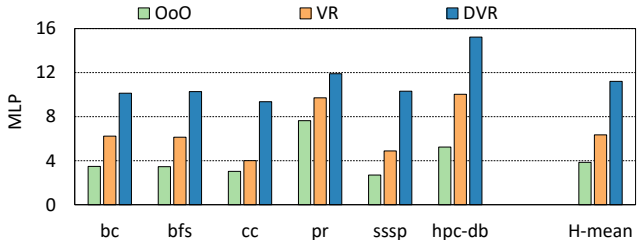

Figure 9: Memory-level parallelism, in terms of MSHRs used per cycle on average, for DVR and VR compared to the baseline OoO core. *DVR generates significantly more parallel outstanding memory accesses.*

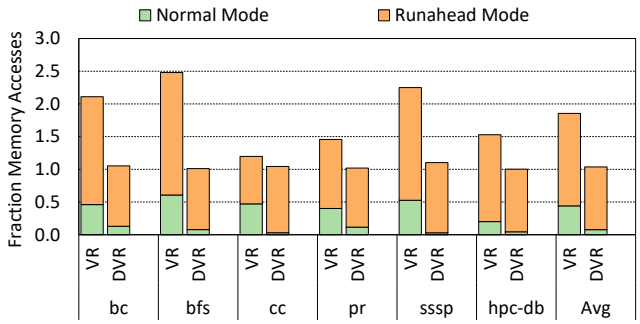

Figure 10: Accuracy and Coverage: number of off-chip memory accesses for VR and DVR normalized to OoO, and fraction of memory accesses in normal versus runahead mode. *DVR successfully prefetches DRAM accesses, converting them into on-chip cache hits when the program subsequently accesses them in normal mode.*

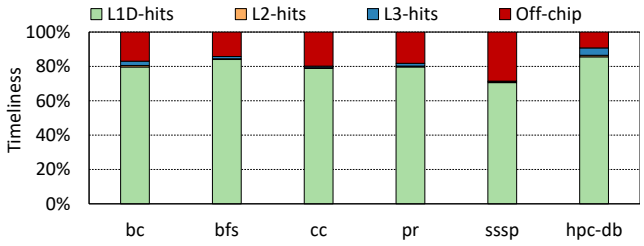

Figure 11: Timeliness: fraction of total prefetched cachelines in runahead mode for which the data is present in the L1-D, L2 and L3 caches during normal mode; 'Off-chip' represents either the cachelines prefetched incorrectly or the cache lines for which the data is still being transferred from memory.

6.3 Memory-Level Parallelism

The secrets of DVR's success are that it generates far more overlapping memory accesses than competing techniques. In Figure 9, we see that the number of outstanding requests for the out-of-order core in the data cache are less than four on average, with DVR generating more than ten at a time, on average per cycle, by comparison. The simplest workloads (pr and those in hpd-db) have fewer branch mispredicts, and so achieve higher raw memory-level parallelism even if the speedups are typically higher in the more complex workloads. Even though DVR itself does not suffer significantly from branch mispredicts (its simple in-order pipeline squashes them extremely early, and its coarse form of speculative loop parallelism means branches across loop iterations do not form chains that cause all instructions later in program order to be squashed), the main thread does, and so DVR naturally ends up looking less far ahead, and overlapping fewer accesses.

6.4 Effectiveness

Here we analyze to what extent DVR is successful at generating accurate, timely, comprehensive prefetches.

Accuracy and Coverage. Figure 10 shows both the total number of main memory accesses performed, and the fraction within the main thread and runahead mode or subthread. Both DVR and VR are given for comparison, relative to the same out-of-order baseline. DVR is extremely accurate because of the Discovery Mode. By contrast, Vector Runahead can over-fetch by over 2×, because it lacks loop-length analysis.

As well as being more accurate, DVR also covers far more of each application, due to triggering more eagerly, and because Nested Mode can handle far more complex indirection.

Timeliness. Figure 11 shows how timely the prefetches are in DVR, in terms of the access latency observed by the main thread. Most cache lines are in the L1 D-cache when the main thread accesses them, with only a few evicted to higher cache levels. This is because the combination of the Discovery and Nested Modes allows DVR to generate very fine-grained memory-level parallelism, meaning that even though we are bringing in hundreds of entries at once, we can synchronize with the main thread so that they are accessed shortly after. Still, a consistent 10–20 percent of accesses observe a latency higher than the last-level cache. When interpreted in correspondence with Figure 10, we see that this is not because of inaccuracy. Rather, it is because the prefetches are too late. Because DVR overlaps with the main thread's execution, and especially because Discovery and Nested Modes can delay the start of vectorization, many earlier accesses in a single runahead iteration may overlap with those same accesses in the main thread. This is a significant (if difficult to avoid) reason why the Oracle, which pays no such overheads for discovering future addresses, achieves better performance in some cases, as previously reported in Figure 7.

6.5 Core Size Sensitivity Analysis

Figure 12 reports performance for DVR as a function of ROB size normalized to our baseline OoO core with 350-entry ROB. In contrast to VR which yields diminishing performance benefits with increasing ROB size, as previously reported in Figure 2, the performance boost offered by DVR holds on. In contrast to VR which is triggered upon a full ROB, DVR operates in a decoupled manner from the main thread, significantly boosting performance by continuously vectorizing and prefetching future chains of dependent loads. When we scale all the back-end structures — in proportion to the ROB — the performance of DVR relative to the OoO baseline

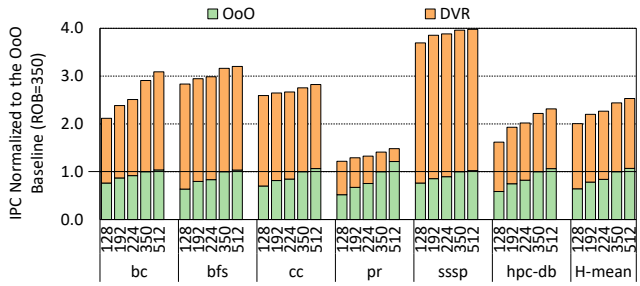

Figure 12: Performance of DVR with increasing ROB size, relative to our baseline OoO core with 350-entry ROB. *The performance gains delivered by DVR continue to increase despite the large size of the ROB.*

with 350-entry ROB is 1.9×, 2.2×, 2.2×, 2.4×, and 2.5× higher for the cores with the ROB sizes of 128, 192, 224, 350, and 512 entries.

7 RELATED WORK

Decoupled Vector Runahead is both a helper thread [20, 44], and a runahead execution [66] according to the categories from Mittal et al. [59] and Falsafi et al. [32].

7.1 Helper Threads and Precomputation

Helper threads perform work to assist the performance of a main thread. Athanasaki et al. [8] perform speculative precomputation on simultaneous multithreads. Wang et al. [93] run helper threads via context switching, removing the need for explicit SMT capabilities. SSMT [20] introduced hardware support specifically for helper threads, rather than SMT generically, and runs microthreads alongside the main core, via a buffer that stores hand-generated micro-ops.

Slice Processors [60] identify cache misses to precompute address calculation on a parallel thread. Dependence-graph computation [7, 82] executes the slices on separate hardware. Lau et al. [53] introduces a small core by not only duplicating the execution hardware, but some of the core front-end as well, to run speculative threads. DeSC [37] decouples address calculation and load-value usage into two separate devices.

Kim et al. [45] generate helper threads automatically in the compiler. Ganusov and Burtscher [35] emulate hardware prefetchers on helper threads. Speculative Precomputation [25] allows helper threads to spawn their own helper threads to handle chain dependencies.

None of this prior work reorders and vectorizes the code in the helper thread to prefetch dependent memory operations far into the future instruction stream, and typically it requires software support instead of being fully microarchitectural.

7.2 Runahead Techniques

Runahead execution [29, 65, 66] was proposed as an alternative to large reorder buffers, allowing execution to continue after a long-latency load by removing the blocking instruction from the reorder buffer and continuing to transiently execute other instructions. Mutlu et al. [62, 64] showed that dynamically choosing whether or not to enter runahead can reduce the number of executed instructions, while keeping the performance benefits intact. Hashemi et al. [40] filter and buffer dependency chains to improve performance. Precise Runahead [70] both filters instructions and avoids throwing away correct instructions that fit inside the ROB. Branch Runahead [78] uses a light dependency chain executed continuously to assist the branch predictor. Bringing runahead together with vectorization was the key idea of Vector Runahead [67, 68] (Section 2.3).

Helper threads have been combined with (scalar) runahead execution [80]. MLP-aware runahead threads [27] only initiate execution with far-distance MLP. Ramirez et al. [79] dynamically calculate the offset at which runahead thread should run. Continuous Runahead [39] offloads simple address patterns to a core at the last-level cache controller, and runs them continuously. As continuous runahead can only prefetch chains leading to independent memory accesses, EMC [38], another near-memory core, prefetches dependent cache misses. Both continuous runahead and EMC are in-order, like DVR, but due to a lack of vectorization and instruction reordering, they cannot deliver high coverage and performance like DVR.

7.3 Auto-Vectorization and SW Reordering

DVR can be seen as a type of speculative vectorization [10, 52, 55, 57, 74, 76, 77, 86], albeit one that is generated microarchitecturally, that does not seek to maintain guaranteed correctness of its transient workload, and which overlaps far more independent loads than a single vector at a time. Likewise, it can be seen as a type of hardware-generated, compute-optimized (via vectorization) software pipelining [21, 47, 81, 90, 92], or software prefetching [3, 4, 17, 21, 61, 89] in that it reorders loads to overlap them.

7.4 Architecturally Visible Prefetching

Prefetching the most complex memory access patterns has traditionally been the preserve of compiler- or hand-targeted hardware. The Event-Triggered Programmable Prefetcher [2] offloads and overlaps many memory accesses like DVR, to hide the latencies of dependent chains. However, it uses compiler- or hand-generated thread-level parallelism, and runs on a sea of small, dedicated cores.

Harbinger instructions [5], Guided-Region Prefetching [94] and RnR [99] generate hints inside programs to give to prefetchers. Prodigy [88] and the Graph Prefetcher [1] are configured with a set of dependent-chain patterns typical to graph workloads. Other prefetchers are configured with the indirection patterns of arrays [19] or linked structures [24, 49]. Such hints may configure the entire memory hierarchy [97].

Fetcher units [41, 48, 50, 51, 56, 73, 100] are configured with the memory access pattern, but directly access data rather than prefetching it, reducing work repetition at the expense of requiring stricter ordering guarantees to preserve correctness.

7.5 Microarchitectural Prefetchers

Stride prefetchers [22, 23], for repeated patterns in addresses such as sequential walks through arrays, are endemic in commercial systems [9]. The recent research literature focuses on improving their coverage, performance and selectivity [11, 15, 46, 58, 71, 84].

More recently, temporal-history prefetchers [42, 43, 72, 95, 96], which store and repeat observed patterns, have become practical enough for deployment in Arm processors [34]. Pythia [14] considers more than just the PC to index predictors, by using reinforcement learning to select the relevant characteristics. Hermes [13] predicts whether data will be cached or off-chip, to avoid waiting for the cache miss before accessing off-chip memory. Shi et. al [85] correlate address patterns via machine learning. The complex data-dependent chains within big-data workloads that DVR targets are unsuited to address correlation, given their lack of regular address pattern, or temporal reuse over even gigabytes of data [2].

Cooksey et al. [26] propose a 'content-directed' prefetcher designed to pick up possible pointers within arrays, with others attempting to reduce overfetch rates via compiler input [5, 30]. IMP is successful at simple indirection patterns [98] but does not scale to graph or database workloads [67]. Takayashiki et al. [87] generate similar simple stride-indirects by observing vector gather instructions. The Bouquet of Prefetchers [75] predicts which prefetcher is best to use for each PC address.

8 CONCLUSION

Decoupled Vector Runahead offloads the runahead execution to a simple, in-order, SIMT, vector subthread that is initiated whenever the core detects an indirect memory access pattern. Unlike prior runahead techniques, DVR does not wait for the reorder buffer to stall, and by discovering the loop bound at runtime, it can adjust the degree of vectorization to better suit application characteristics. DVR generates prefetches from multiple invocations of a loop when the discovered degree of vectorization for one invocation is not sufficient to achieve high memory-level parallelism. DVR incurs minimal hardware overhead of 1139 bytes.

The benefits of reordering-based runahead over invalidation runaheads will usher in a new era of processors with the latency insensitivity of GPUs while maintaining the programmability and single-threaded performance of CPUs. The potential of near-oracle performance for even the trickiest graph workloads is too tempting to leave on the table.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable feedback. This work is supported in part by the UGent-BOF-GOA grant No. 01G01421, the Research Foundation Flanders (FWO) grant No. G018722N, the 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] Sam Ainsworth and Timothy M. Jones. 2016. Graph Prefetching Using Data Structure Knowledge. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 39, 11 pages. https://doi.org/10.1145/2925426.2926254

[2] Sam Ainsworth and Timothy M. Jones. 2018. 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* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 578–592. https://doi.org/10.1145/3173162.3173189

[3] Sam Ainsworth and Timothy M. Jones. 2019. Software Prefetching for Indirect Memory Accesses: A Microarchitectural Perspective. *ACM Transactions on Computer Systems* 36, 3, Article 8 (jun 2019), 34 pages. https://doi.org/10.1145/3319393

[4] Sam Ainsworth and Timothy M. Jones. 2020. Prefetching in Functional Languages. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management* (London, UK) *(ISMM '20)*. Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3381898.3397209

[5] Hassan Al-Sukhni, Ian Bratt, and Daniel A. Connors. 2003. Compiler-Directed Content-Aware Prefetching for Dynamic Data Structures. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*. IEEE Computer Society, Los Alamitos, CA, USA, 91. https://doi.org/10.1109/PACT.2003.1238005

[6] James Alfred Ang, Brian W. Barrett, Kyle Bruce Wheeler, and Richard C. Murphy. 2010. Introducing the graph 500. *Cray User's Group (CUG)* 19 (5 2010), 45–74. https://www.osti.gov/biblio/1014641

[7] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. 2001. Data Prefetching by Dependence Graph Precomputation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture* (Göteborg, Sweden) *(ISCA '01)*. Association for Computing Machinery, New York, NY, USA, 52–61. https://doi.org/10.1145/379240.379251

[8] Evangelia Athanasaki, Nikos Anastopoulos, Kornilios Kourtis, and Nectarios Koziris. 2008. Exploring the Performance Limits of Simultaneous Multithreading for Memory Intensive Applications. *Journal of Supercomputing* 44, 1 (apr 2008), 64–97. https://doi.org/10.1007/s11227-007-0149-x

[9] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 513–526. https://doi.org/10.1145/3373376.3378498

[10] Sara S. Baghsorkhi, Nalini Vasudevan, and Youfeng Wu. 2016. FlexVec: Auto-Vectorization for Irregular Loops. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 697–710. https://doi.org/10.1145/2908080.2908111

[11] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo Spatial Data Prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA '19)*. IEEE Computer Society, Los Alamitos, CA, USA, 399–411. https://doi.org/10.1109/HPCA.2019.00053

[12] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[13] Rahul Bera, Konstantinos Kanellopoulos, Shankar Balachandran, David Novo, Ataberk Olgun, Mohammad Sadrosadat, and Onur Mutlu. 2022. Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO-55)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–18. https://doi.org/10.1109/MICRO56248.2022.00015

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

[15] Rahul Bera, Anant V. Nori, Onur Mutlu, and Sreenivas Subramoney. 2019. DSPatch: Dual Spatial Pattern Prefetcher. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 531–544. https://doi.org/10.1145/3352460.3358325

[16] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology* 25, 2 (2001), 163–177. https://doi.org/10.1080/0022250X.2001.9990249

[17] David Callahan, Ken Kennedy, and Allan Porterfield. 1991. Software Prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, USA) *(ASPLOS IV)*. Association for Computing Machinery, New York, NY, USA, 40–52. https://doi.org/10.1145/106972.106979

[18] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization* 11, 3, Article 28 (aug 2014), 25 pages. https://doi.org/10.1145/2629677

[19] Mustafa Cavus, Resit Sendag, and Joshua J. Yi. 2020. Informed Prefetching for Indirect Memory Accesses. *ACM Transactions on Architecture and Code Optimization* 17, 1, Article 4 (mar 2020), 29 pages. https://doi.org/10.1145/3374216

[20] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. 1999. Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture* (Atlanta, Georgia, USA) *(ISCA '99)*. IEEE Computer Society, Los Alamitos, CA, USA, 186–195. https://doi.org/10.1145/300979.300995

[21] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving Hash Join Performance through Prefetching. *ACM Transactions on Database Systems* 32, 3 (aug 2007), 17–es. https://doi.org/10.1145/1272743.1272747

[22] Tien-Fu Chen and Jean-Loup Baer. 1992. Reducing Memory Latency via Non-blocking and Prefetching Caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS V)*. Association for Computing Machinery, New York, NY, USA, 51–61. https://doi.org/10.1145/143365.143486

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

[24] Seungryul Choi, Nicholas Kohout, Sumit Pamnani, Dongkeun Kim, and Donald Yeung. 2004. A General Framework for Prefetch Scheduling in Linked Data Structures and Its Application to Multi-chain Prefetching. *ACM Transactions on Computer Systems* 22, 2 (may 2004), 214–280. https://doi.org/10.1145/986533.986536

[25] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. 2001. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture* (Göteborg, Sweden) *(ISCA '01)*. Association for Computing Machinery, New York, NY, USA, 14–25. https://doi.org/10.1145/379240.379248

[26] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. 2002. A Stateless, Content-directed Data Prefetching Mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 279–290. https://doi.org/10.1145/605397.605427

[27] Kenzo Van Craeynest, Stijn Eyerman, and Lieven Eeckhout. 2009. MLP-Aware Runahead Threads in a Simultaneous Multithreading Processor. In *High Performance Embedded Architectures and Compilers, Fourth International Conference, HiPEAC 2009, Paphos, Cyprus, January 25-28, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5409)*. Springer Berlin Heidelberg, Berlin, Heidelberg, 110–124. https://doi.org/10.1007/978-3-540-92990-1_10

[28] Dr. Ian Cutress. 2018. *Intel's Architecture Day 2018: The future of core, Intel gpus, 10nm, and hybrid x86*. AnandTech. https://www.anandtech.com/show/13699/intel-architecture-day-2018-core-future-hybrid-x86

[29] James Dundas and Trevor Mudge. 1997. Improving Data Cache Performance by Pre-Executing Instructions under a Cache Miss. In *Proceedings of the 11th International Conference on Supercomputing* (Vienna, Austria) *(ICS '97)*. Association for Computing Machinery, New York, NY, USA, 68–75. https://doi.org/10.1145/263580.263597

[30] Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, USA, 7–17. https://doi.org/10.1109/HPCA.2009.4798232

[31] Jack Edmonds and Richard M. Karp. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM* 19, 2 (April 1972), 248–264. https://doi.org/10.1145/321694.321699

[32] Babak Falsafi and Thomas F. Wenisch. 2014. *A Primer on Hardware Prefetching*. Springer Cham, Cham, Switzerland. https://doi.org/10.1007/978-3-031-01743-8

[33] Andrei Frumusanu. 2020. *Apple Announces The Apple Silicon M1: Ditching x86 - What to Expect, Based on A14*. Anandtech. https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2

[34] Andrei Frumusanu. 2021. *The Snapdragon 888 vs The Exynos 2100: Cortex-X1 & 5nm - Who Does It Better?* AnandTech. https://www.anandtech.com/show/16463/snapdragon-888-vs-exynos-2100-galaxy-s21-ultra/3

[35] Ilya Ganusov and Martin Burtscher. 2006. Efficient Emulation of Hardware Prefetchers via Event-Driven Helper Threading. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques* (Seattle, Washington, USA) *(PACT '06)*. Association for Computing Machinery, New York, NY, USA, 144–153. https://doi.org/10.1145/1152154.1152178

[36] Saurabh Gupta, Niranjan Soundararajan, Ragavendra Natarajan, and Sreenivas Subramoney. 2020. Opportunistic Early Pipeline Re-Steering for Data-Dependent Branches. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/3410463.3414628

[37] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled Supply-compute Communication Management for Heterogeneous Architectures. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 191–203. https://doi.org/10.1145/2830772.2830800

[38] Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2016. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Computer Society, Los Alamitos, CA, USA, 444–455. https://doi.org/10.1109/ISCA.2016.46

[39] Milad Hashemi, Onur Mutlu, and Yale N. Patt. 2016. Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Computer Society, Los Alamitos, CA, USA, Article 61, 12 pages. https://doi.org/10.1109/MICRO.2016.7783764

[40] Milad Hashemi and Yale N. Patt. 2015. Filtered Runahead Execution with a Runahead Buffer. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 358–369. https://doi.org/10.1145/2830772.2830812

[41] Chen-Han Ho, Sung Jin Kim, and Karthikeyan Sankaralingam. 2015. Efficient Execution of Memory Access Phases Using Dataflow Specialization. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 118–130. https://doi.org/10.1145/2749469.2750390

[42] Akanksha Jain and Calvin Lin. 2013. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 247–259. https://doi.org/10.1145/2540708.2540730

[43] Doug Joseph and Dirk Grunwald. 1997. Prefetching Using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (Denver, Colorado, USA) *(ISCA '97)*. Association for Computing Machinery, New York, NY, USA, 252–263. https://doi.org/10.1145/264107.264207

[44] Changhee Jung, Daeseob Lim, Jaejin Lee, and Yan Solihin. 2006. Helper Thread Prefetching for Loosely-Coupled Multiprocessor Systems. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing* (Rhodes Island, Greece) *(IPDPS'06)*. IEEE Computer Society, Los Alamitos, CA, USA, 10 pp.–. https://doi.org/10.1109/IPDPS.2006.1639375

[45] Dongkeun Kim and Donald Yeung. 2002. Design and Evaluation of Compiler Algorithms for Pre-execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 159–170. https://doi.org/10.1145/605397.605415

[46] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A. L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path Confidence Based Lookahead Prefetching. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Computer Society, Los Alamitos, CA, USA, Article 60, 12 pages. https://doi.org/10.1109/MICRO.2016.7783763

[47] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *Proc. VLDB Endow.* 9, 4 (dec 2015), 252–263. https://doi.org/10.14778/2856318.2856321

[48] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. Association for Computing Machinery, New York, NY, USA, 468–479. https://doi.org/10.1145/2540708.2540748

[49] Nicholas Kohout, Seungryul Choi, Dongkeun Kim, and Donald Yeung. 2001. Multi-Chain Prefetching: Effective Exploitation of Inter-Chain Memory Parallelism for Pointer-Chasing Codes. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT '01)*. IEEE Computer Society, Los Alamitos, CA, USA, 268–279. https://doi.org/10.1109/PACT.2001.953307

[50] Snehasish Kumar, Arrvindh Shriraman, Vijayalakshmi Srinivasan, Dan Lin, and Jordon Phillips. 2014. SQRL: Hardware Accelerator for Collecting Software Data Structures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 475–476. https://doi.org/10.1145/2628071.2628118

[51] Snehasish Kumar, Naveen Vedula, Arrvindh Shriraman, and Vijayalakshmi Srinivasan. 2015. DASX: Hardware Accelerator for Software Data Structures. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 361–372. https://doi.org/10.1145/2751205.2751231

[52] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

[53] Eric Lau, Jason E. Miller, Inseok Choi, Donald Yeung, Saman Amarasinghe, and Anant Agarwal. 2011. Multicore Performance Optimization Using Partner Cores. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*. USENIX

Association, Berkeley, CA, 1–6. https://www.usenix.org/conference/hotpar11/multicore-performance-optimization-using-partner-cores

[54] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro* 28, 2 (March 2008), 39–55. https://doi.org/10.1109/MM.2008.31

[55] Jun Liu, Yuanrui Zhang, Ohyoung Jang, Wei Ding, and Mahmut Kandemir. 2012. A Compiler Framework for Extracting Superword Level Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 347–358. https://doi.org/10.1145/2254064.2254106

[56] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. 2020. Livia: Data-Centric Computing Throughout the Memory Hierarchy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 417–433. https://doi.org/10.1145/3373376.3378497

[57] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE Computer Society, Los Alamitos, CA, USA, 372–382. https://doi.org/10.1109/PACT.2011.68

[58] Pierre Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 469–480. https://doi.org/10.1109/HPCA.2016.7446087

[59] Sparsh Mittal. 2016. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Comput. Surv.* 49, 2, Article 35 (aug 2016), 35 pages. https://doi.org/10.1145/2907071

[60] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi. 2001. Slice-Processors: An Implementation of Operation-Based Prediction. In *Proceedings of the 15th International Conference on Supercomputing* (Sorrento, Italy) *(ICS '01)*. Association for Computing Machinery, New York, NY, USA, 321–334. https://doi.org/10.1145/377792.377856

[61] Todd Carl Mowry. 1995. *Tolerating Latency through Software-Controlled Data Prefetching*. Ph. D. Dissertation. Stanford University, Computer Systems Laboratory, Stanford, CA, USA. UMI Order No. GAX94-29983.

[62] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2005. Techniques for Efficient Processing in Runahead Execution Engines. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*. IEEE Computer Society, Los Alamitos, CA, USA, 370–381. https://doi.org/10.1109/ISCA.2005.49

[63] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2006. Address-Value Delta (AVD) Prediction: A Hardware Technique for Efficiently Parallelizing Dependent Cache Misses. *IEEE Trans. Comput.* 55, 12 (Dec 2006), 1491–1508. https://doi.org/10.1109/TC.2006.191

[64] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2006. Efficient Runahead Execution: Power-Efficient Memory Latency Tolerance. *IEEE Micro* 26, 1 (Jan 2006), 10–20. https://doi.org/10.1109/MM.2006.10

[65] Onur Mutlu, Hyesoon Kim, Jared Stark, and Yale N. Patt. 2005. On Reusing the Results of Pre-Executed Instructions in a Runahead Execution Processor. *IEEE Computer Architecture Letters* 4, 1 (Jan 2005), 2–2. https://doi.org/10.1109/L-CA.2005.1

[66] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE Computer Society, Los Alamitos, CA, USA, 129–140. https://doi.org/10.1109/HPCA.2003.1183532

[67] Ajeya Naithani, Sam Ainsworth, Timothy M. Jones, and Lieven Eeckhout. 2021. Vector Runahead. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Computer Society, Los Alamitos, CA, USA, 195–208. https://doi.org/10.1109/ISCA52012.2021.00024

[68] Ajeya Naithani, Sam Ainsworth, Timothy M. Jones, and Lieven Eeckhout. 2022. Vector Runahead for Indirect Memory Accesses. *IEEE Micro* 42, 4 (jul 2022), 116–123. https://doi.org/10.1109/MM.2022.3163132

[69] Ajeya Naithani, Josué Feliu, Almutaz Adileh, and Lieven Eeckhout. 2019. Precise Runahead Execution. *IEEE Computer Architecture Letters* 18, 1 (Jan 2019), 71–74. https://doi.org/10.1109/LCA.2019.2910518

[70] Ajeya Naithani, Josué Feliu, Almutaz Adileh, and Lieven Eeckhout. 2020. Precise Runahead Execution. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 397–410. https://doi.org/10.1109/HPCA47549.2020.00040

[71] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals-Yúfera, and Alberto Ros. 2022. Berti: an Accurate Local-Delta Data Prefetcher. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO-55)*. IEEE Computer Society, Los Alamitos, CA, USA, 975–991. https://doi.org/10.1109/MICRO56248.2022.00072

[72] Kyle J. Nesbit and James E. Smith. 2004. Data Cache Prefetching Using a Global History Buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA '04)*. IEEE Computer Society, Los Alamitos, CA, USA, 96. https://doi.org/10.1109/HPCA.2004.10030

[73] Quan M. Nguyen and Daniel Sanchez. 2020. Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 596–608. https://doi.org/10.1109/MICRO50266.2020.00056

[74] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-Vectorization of Interleaved Data for SIMD. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 132–143. https://doi.org/10.1145/1133981.1133997

[75] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 118–131. https://doi.org/10.1109/ISCA45697.2020.00021

[76] Vasileios Porpodas and Timothy M. Jones. 2015. Throttling Automatic Vectorization: When Less is More. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Los Alamitos, CA, USA, 432–444. https://doi.org/10.1109/PACT.2015.32

[77] Vasileios Porpodas, Alberto Magni, and Timothy M. Jones. 2015. PSLP: Padded SLP Automatic Vectorization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (San Francisco, California) *(CGO '15)*. IEEE Computer Society, Los Alamitos, CA, USA, 190–201. https://doi.org/10.1109/CGO.2015.7054199

[78] Stephen Pruett and Yale Patt. 2021. Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 804–815. https://doi.org/10.1145/3466752.3480053

[79] Tanausú Ramírez, Alex Pajuelo, Oliverio Jesus Santana, Onur Mutlu, and Mateo Valero. 2010. Efficient Runahead Threads. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (Vienna, Austria) *(PACT '10)*. Association for Computing Machinery, New York, NY, USA, 443–452. https://doi.org/10.1145/1854273.1854328

[80] Tanausú Ramírez, Alex Pajuelo, Oliverio Jesus Santana, and Mateo Valero. 2008. Runahead Threads to improve SMT performance. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, USA, 149–158. https://doi.org/10.1109/HPCA.2008.4658635

[81] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled Software Pipelining with the Synchronization Array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, Los Alamitos, CA, USA, 177–188. https://doi.org/10.1109/PACT.2004.1342552

[82] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. 1998. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VIII)*. Association for Computing Machinery, New York, NY, USA, 115–126. https://doi.org/10.1145/291069.291034

[83] André Seznec. 2016. TAGE-SC-L Branch Predictors Again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)* (Seoul, South Korea). INRIA HAL, rennes France, 1–4. https://inria.hal.science/hal-01354253

[84] Manjunath Shevgoor, Sahil Koladiya, Rajeev Balasubramonian, Chris Wilkerson, Seth H. Pugsley, and Zeshan Chishti. 2015. Efficiently Prefetching Complex Address Patterns. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 141–152. https://doi.org/10.1145/2830772.2830793

[85] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A Hierarchical Neural Model of Data Prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 861–873. https://doi.org/10.1145/3445814.3446752

[86] Peng Sun, Giacomo Gabrielli, and Timothy M. Jones. 2021. Speculative Vectorisation with Selective Replay. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Computer Society, Los Alamitos, CA, USA, 223–236. https://doi.org/10.1109/ISCA52012.2021.00026

[87] Hikaru Takayashiki, Masayuki Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. 2019. A Hardware Prefetching Mechanism for Vector Gather Instructions. In *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE Computer Society, Los Alamitos, CA, USA, 59–66. https://doi.org/10.1109/IA349570.2019.00015

[88] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor

Mudge, and Ronald Dreslinski. 2021. 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. *Proceedings - International Symposium on High-Performance Computer Architecture* 2021-February, 654–667. https://doi.org/10.1109/HPCA51647.2021.00061

[89] Sam Ainsworth Timothy and M. Jones. 2017. Software prefetching for indirect memory accesses. In *CGO 2017 - Proceedings of the 2017 International Symposium on Code Generation and Optimization*. IEEE Computer Society, Los Alamitos, CA, USA, 305–317. https://doi.org/10.1109/CGO.2017.7863749

[90] Kim-Anh Tran, Trevor E. Carlson, Konstantinos Koukos, Magnus Själander, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2017. Clairvoyance: Look-Ahead Compile-Time Scheduling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Computer Society, Los Alamitos, CA, USA, 171–184. https://doi.org/10.1109/CGO.2017.7863738

[91] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous Multithreading: Maximizing on-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (S. Margherita Ligure, Italy) *(ISCA '95)*. Association for Computing Machinery, New York, NY, USA, 392–403. https://doi.org/10.1145/223982.224449

[92] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. 2007. Speculative Decoupled Software Pipelining. In *2007 16th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Los Alamitos, CA, USA, 49–59. https://doi.org/10.1109/PACT.2007.66

[93] Perry H. Wang, Jamison D. Collins, Hong Wang, Dongkeun Kim, Bill Greene, Kai-Ming Chan, Aamir B. Yunus, Terry Sych, Stephen F. Moore, and John P. Shen. 2004. Helper Threads via Virtual Multithreading. *IEEE Micro* 24, 6 (nov 2004), 74–82. https://doi.org/10.1109/MM.2004.75

[94] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. 2003. Guided Region Prefetching: A Cooperative Hardware/Software Approach. In *Proceedings of the 30th Annual International Symposium on Computer Architecture* (San Diego, California) *(ISCA '03)*. Association for Computing Machinery, New York, NY, USA, 388–398. https://doi.org/10.1145/859618.859663

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

[96] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Efficient Metadata Management for Irregular Data Prefetching. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 449–461. https://doi.org/10.1145/3307650.3322225

[97] Chia-Lin Yang and Alvin R. Lebeck. 2002. A Programmable Memory Hierarchy for Prefetching Linked Data Structures. In *Proceedings of the 4th International Symposium on High Performance Computing (ISHPC '02)*. Springer-Verlag, Berlin, Heidelberg, 160–174. https://doi.org/10.1007/3-540-47847-7_15

[98] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect Memory Prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 178–190. https://doi.org/10.1145/2830772.2830807

[99] Chao Zhang, Yuan Zeng, John Shalf, and Xiaochen Guo. 2020. RnR: A Software-Assisted Record-and-Replay Hardware Prefetcher. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 609–621. https://doi.org/10.1109/MICRO50266.2020.00057

[100] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 593–607. https://doi.org/10.1145/3173162.3173197