Vector Runahead

Ajeya Naithani[†] Sam Ainsworth[‡] Timothy M. Jones[≀] Lieven Eeckhout[†]

[†]Ghent University [‡]University of Edinburgh [≀]University of Cambridge

***Abstract*—The memory wall places a significant limit on performance for many modern workloads. These applications feature complex chains of dependent, indirect memory accesses, which cannot be picked up by even the most advanced microarchitectural prefetchers. The result is that current out-of-order superscalar processors spend the majority of their time stalled. While it is possible to build special-purpose architectures to exploit the fundamental memory-level parallelism, a microarchitectural technique to automatically improve their performance in conventional processors has remained elusive.**

Runahead execution is a tempting proposition for hiding latency in program execution. However, to achieve high memory-level parallelism, a standard runahead execution skips ahead of cache misses. In modern workloads, this means it only prefetches the first cache-missing load in each dependent chain. We argue that this is not a fundamental limitation. If runahead were instead to stall on cache misses to generate dependent chain loads, then it could regain performance if it could stall on many at once. With this insight, we present Vector Runahead, a technique that prefetches entire load chains and speculatively reorders scalar operations from multiple loop iterations into vector format to bring in many independent loads at once. Vectorization of the runahead instruction stream increases the effective fetch/decode bandwidth with reduced resource requirements, to achieve high degrees of memory-level parallelism at a much faster rate. Across a variety of memory-latency-bound indirect workloads, Vector Runahead achieves a 1.79× performance speedup on a large out-of-order superscalar system, significantly improving on state-of-the-art techniques.

I. Introduction

Modern-day workloads are poorly served by current out-of-order superscalar cores. From databases [40], to graph workloads [49, 67], to HPC codes [11, 30], many workloads feature sparse, indirect memory accesses [9] characterized by high-latency cache misses that are unpredictable by today's stride prefetchers [1, 19]. For these workloads, even out-of-order superscalar processors spend the majority of their time stalled, since their ample reorder buffer and issue queue resources are still insufficient to capture the memory-level parallelism necessary to hide today's DRAM latencies.

Still, this performance gap is not insurmountable. Many elaborate accelerators [1, 3, 40, 44, 45, 48] have shown significant success in improving the performance of these workloads through domain-specific programming models and/or specialized hardware units dedicated to the task of fetching in new data. Part of the gap can even be eliminated with sophisticated programmer- and compiler-directed software-prefetching mechanisms [2, 4, 30, 41].

More ideal is a pure-microarchitecture technique that can do the same: achieving high performance for memory-latency-bound workloads, while requiring no programmer input, and being binary-compatible with today's applications. For simple stride patterns, such techniques are endemic in today's cache systems [19]. For more complex indirection patterns, the inability at the cache-system level to identify complex load chains and generate their addresses limits existing techniques to simple array-indirect [88] and pointer-chasing [23] codes.

To achieve the instruction-level visibility necessary to calculate the addresses of complex access patterns seen in today's workloads [3], we conclude that this ideal technique must operate within the core, instead of within the cache. Runahead execution [25, 32, 34, 57, 58, 64] is the most promising technique to date, where upon a memory stall at the head of the reorder buffer (ROB), execution enters a speculative 'runahead' mode designed to prefetch future memory accesses. In runahead mode, the addresses of future memory accesses are calculated and the memory accesses are speculatively issued. When the blocking load miss returns, the ROB unblocks and the processor resumes normal-mode execution, at which time the memory accesses hit in the nearby caches. Runahead execution is a highly effective technique to identify independent load misses in the future instruction stream when the processor is stalled on a first long-latency load miss. By speculatively issuing multiple independent memory accesses, runahead execution significantly increases memory-level parallelism (MLP), ultimately improving overall application performance.

While runahead execution successfully prefetches independent load misses in the future instruction stream, it suffers from three fundamental limitations. First, runahead is unsuitable for complex indirection patterns that consist of chains of dependent load misses [62]. While combining a traditional hardware stride prefetcher with runahead might enable prefetching one level of indirection, it does not provide a general solution to prefetch all loads in a chain of dependent loads. Second, runahead execution is limited by the processor's front-end (fetch/decode/rename) width: the rate at which runahead execution can generate MLP is slow if there is a large number of instructions between the independent loads in the future instruction stream. Third, the speculation depth of runahead is limited by the amount of available back-end resources (issue queue slots and physical registers) [64].

In this paper, we propose *Vector Runahead*, a novel runahead technique that overcomes the above limitations through three key innovations. First, Vector Runahead alters the runahead's termination condition by remaining in runahead mode until all loads in the dependence chain have been issued, as opposed to returning to normal mode as soon as the blocking load miss returns from main memory. This enables Vector Runahead to prefetch entire load chains. Second,

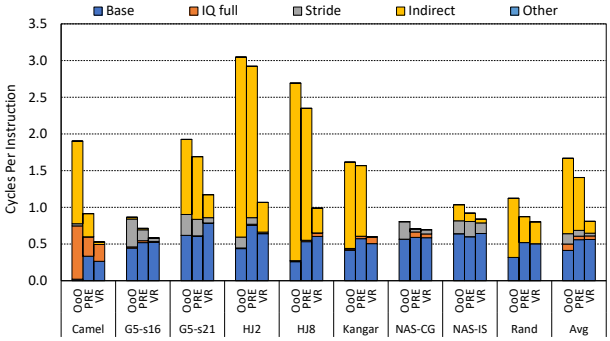


Fig. 1: CPI stacks for an out-of-order (OoO) core, Precise Runahead Execution (PRE) and our new Vector Runahead (VR). The memory component is broken down and attributed to striding loads and indirect dependent-chain loads. *The previous state-of-the-art runahead cannot prefetch the majority of (indirect) memory accesses, unlike Vector Runahead.*

Vector Runahead vectorizes the runahead instruction stream by reinterpreting scalar instructions as vector operations to generate many different cache misses at different offsets. It subsequently gathers many dependent loads in the instruction sequence, hiding cache latency even for complex memory-access patterns. In effect, vectorization virtually increases the effective fetch/decode bandwidth during runahead mode, while at the same time requiring very few back-end resources, e.g., only a single issue queue slot is required for a vector operation with 8 (memory) operations. Third, Vector Runahead issues multiple rounds of these vectorized instructions through vector unrolling and pipelining to speculate even deeper and increase the effective runahead fetch/decode bandwidth even further — e.g., 8 rounds of vector runahead with 8 vector loads each, lead to 64 speculative prefetches that are issued in parallel.

We evaluate Vector Runahead through detailed simulation using a variety of graph, database and HPC workloads, and we report that Vector Runahead improves performance by 1.79× compared to a baseline out-of-order processor — a significant improvement over the state-of-the-art Precise Runahead Execution (PRE) technique [64] which achieves a speedup of 1.20×. The performance speedup results from much higher memory-level parallelism by prefetching loads within dependent load sequences in an accurate and timely manner. Vector Runahead does not significantly impact system complexity, adding only 1.3 KB of new state over our baseline.

II. Background and Motivation

Before explaining Vector Runahead in detail, we first provide context around the limitations of previous runahead techniques, and the potential left to be exploited.

A. Memory Stalls in Out-of-Order Cores

Modern out-of-order (OoO) cores frequently stall on long-latency memory accesses. When a load turns out to be a last-level cache miss, it often reaches the head of the reorder buffer (ROB), stalling commit as it waits for the miss to return. In the meantime, the front-end pipeline continues dispatching instructions until the ROB completely fills up, leading to a *full-window stall*. Such a memory access typically stalls the core for tens to hundreds of cycles. Figure 1 shows CPI stacks for a range of benchmarks on an OoO core [27] (see Section V for our experimental setup). In addition to the cycles spent on performing useful work (shown as the 'Base' component), the CPI stacks also show the number of cycles the processor is waiting due to a full issue queue ('IQ full') or a memory access. Memory-access cycles are divided into three types of load instruction stalling the processor: (1) striding load instructions ('Stride'); (2) indirect load instructions that directly or indirectly depend on a striding load instruction ('Indirect'); and (3) other types of load instructions ('Other'). We find that indirect load instructions stall the processor for 61.6% of the total execution time on average, and up to 89.8% (HJ8). For high performance, it is critical to eliminate stalls from indirect memory accesses in OoO cores.

B. Limitations of Runahead Techniques

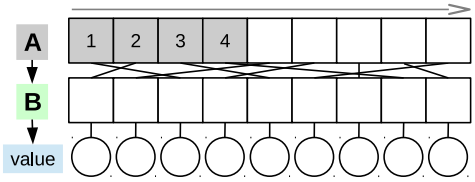
To alleviate the bottleneck caused by memory accesses, standard runahead execution [25, 32, 34, 57, 58] checkpoints and releases the architectural state of an application after a full-window stall and enters *runahead* mode. The processor then continues speculatively generating memory accesses. When the blocking memory access returns, the runahead *interval* terminates, at which point the pipeline is flushed, the architectural state is restored to the point of entry to runahead, and *normal* execution resumes. The prefetches generated during runahead mode bring future data into the processor caches that reduce the number of upcoming stalls during normal mode, thus improving performance.

Precise Runahead Execution (PRE) [63, 64], the state-of-the-art in runahead execution, improves upon standard runahead through three key mechanisms. (1) PRE leverages the available back-end (issue queue and physical register file) resources to speculatively execute instructions in runahead mode, thereby eliminating the need to release and flush processor state when entering and exiting runahead mode. (2) PRE only speculatively pre-executes instructions that are required to generate memory accesses after a full-window stall. (3) PRE includes a mechanism to quickly recycle back-end resources during runahead mode. PRE's performance benefits come from reduced overheads when transitioning between runahead mode and normal mode, which enables running ahead even during short runahead intervals, and only dispatching loads and their address-generating instructions during runahead mode, and not the dependents of memory accesses (as long as they do not lead to dependent loads), thereby reducing the amount of back-end resources needed during runahead mode.

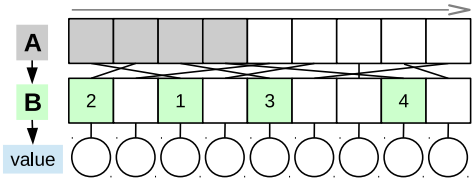
PRE prefetches a substantial fraction of the memory accesses in the upcoming instruction stream, i.e., PRE reduces the total fraction of processor cycles stalled on memory accesses by 31.5% on average, see Figure 1. Although PRE eliminates some of the full-ROB stalls caused by indirect memory accesses, it fails to prefetch the majority of the indirect memory accesses. Accesses beyond the reach of the runahead interval soon stall the core again, causing the pro-

```
for (int x=0; x<N; x++)
    y += B[hash(A[x])]->value;
```

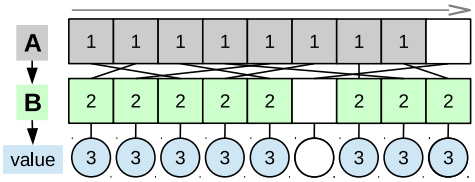
(a) Example code, with memory access by array indirection, with intermediate address computation and pointer access.



(b) PRE is able to prefetch array elements from A. In contrast, the array elements to B cannot be prefetched during runahead mode as they depend on A. Likewise, the data values cannot be prefetched either because they depend on B. Note that the elements in A are accessed serially as indicated. PRE runahead mode is terminated before it can prefetch array elements of B; furthermore, the number of back-end resources needed during runahead mode limits the speculation depth.



(c) Hardware stride prefetching can prefetch the array elements from A, enabling PRE to prefetch the array elements from B during runahead mode. Unfortunately, the data values cannot be prefetched during runahead mode as they depend on B. Note that the order in which the elements in B are indexed depends on the pointer values in A. The same limitations apply as in Figure 2(b) off-by-one.



(d) Vector Runahead vectorizes memory accesses along the memory dependence chain during runahead mode. Multiple accesses to A happen in parallel, followed by parallel accesses to B, followed by parallel data value reads.

Fig. 2: Vector Runahead versus PRE, with and without stride prefetching, for an illustrative code example. The loads highlighted in green can only be triggered by stalling on loads highlighted in gray, and those in blue by stalling on gray and green. *Vector Runahead prefetches multiple memory accesses in parallel along the memory dependence chain during runahead mode.*

cessor to re-enter runahead mode. Consequently, the processor remains stalled for 51.2% of the total execution time on indirect memory accesses on average, and up to 76.5% (HJ8). Overall, despite a positive performance impact, PRE still leaves huge potential for improving performance because of its inability to effectively prefetch indirect memory accesses.

C. Vector Runahead Execution for Dependent Loads

We now analyze in more detail why PRE fails to effectively prefetch indirect memory accesses using the illustrative example shown in Figure 2(a). This code example includes two levels of indirection: the index to array A leads to a (hashed) index into array B, which in turn leads to a dependent access of a data value.

PRE is initiated upon a full-window stall, and speculatively executes the future instruction stream during runahead mode. As illustrated in Figure 2(b) and assuming all memory accesses miss in the cache, PRE will issue a prefetch for the access to the first element in array A. The instructions that depend on this memory access cannot execute, and hence the access to array B and the pointer value cannot be prefetched. As the processor continues through the instruction stream during runahead mode, it will hit the access to the second element in array A, for which it will issue another prefetch; unfortunately, the dependent accesses to B and the data values cannot be prefetched. Next, a prefetch is issued for the third element in A, etc. Runahead mode stops when the blocking load that initiated runahead mode returns from main memory. At this point, a number of accesses to array A have been prefetched but not the elements to array B nor the dependent data values. Once back in normal mode, the processor will soon stall again due to the first cache miss to array B.

Modern-day processors typically feature a hardware stride prefetcher that should be able to prefetch the strided accesses to array A. If so, when PRE accesses elements from array A, they will hit in the cache, and therefore PRE is able to issue prefetch requests for the first level of indirection to array B, as illustrated in Figure 2(c). During runahead mode, PRE will turn the first access to array B (labeled '1') into a prefetch request. The dependent data value cannot be prefetched though as it depends on the access to B. The next access to A leads to a hit in the cache — as it was successfully prefetched by the stride prefetcher — and hence the processor can compute the address for the next access in array B and issue a prefetch request (labeled '2'). Again, the dependent data value cannot be prefetched. The next access to A is a cache hit, and the processor then prefetches the third access to B (labeled '3'), etc. In summary, even though stride prefetching enables runahead execution to prefetch one more level of indirection, it still cannot prefetch the second level (or beyond).

The fact that PRE can prefetch future memory accesses while being stalled on a first initiating memory access, improves MLP and overall performance. Unfortunately, as illustrated in this example, PRE runs into a number of limitations. First, PRE is unable to prefetch all loads along a chain of dependent loads because the data required for the next access in the chain is not available. A hardware prefetcher alongside runahead solves the problem off-by-one, but does not provide a general solution. Second, the rate at which PRE can issue speculative prefetches is limited. Indeed, while runahead is able to service some independent loads from multiple load chains simultaneously, it takes time before those independent loads can be issued, limiting the effective MLP that can be achieved — this is especially the case if the number of instructions per loop iteration is large, e.g., a complicated hash function in the example in Figure 2(a). Third, we note that a

significant number of processor back-end resources are needed during runahead mode to reach the independent loads, which can limit PRE's speculation depth into the future.

Vector Runahead overcomes these fundamental limitations as follows. Vector Runahead changes runahead mode's termination condition, i.e., instead of returning to normal mode once the blocking load miss returns from main memory, Vector Runahead continues runahead mode until all loads along the dependent load chain have been issued. In addition, Vector Runahead vectorizes the dynamic instruction stream during runahead mode, which in effect is equivalent to running ahead at a much faster rate. This is illustrated in Figure 2(d). When vector-runahead mode is initiated, multiple accesses to array A are vectorized, i.e., the same memory operation is speculatively issued at multiple induction-variable offsets in parallel. We vectorize as many copies as the available vector width, or 8 in this example. Instructions that depend on the values from array A are also vectorized, including the accesses to array B and the dependent data values. Vectorizing the dependent instruction stream during runahead mode, while staying in runahead mode until the last dependent load has been issued, enables speculatively prefetching the entire chain of dependent loads. Vectorizing the runahead instruction stream has the effect of issuing the same memory operations from multiple iterations of the loop simultaneously, before issuing the next parallel batch of dependent memory accesses, etc. This effective reordering of memory accesses compared to the original instruction stream enables Vector Runahead to first issue a batch of accesses to A, then a batch of accesses to B, and finally a batch of accesses to the dependent data values. In other words, even though independent memory accesses may be far apart from each other in the original dynamic instruction stream, Vector Runahead issues them in parallel. This reordering of memory accesses is enabled through vectorization which features two key benefits: (1) it substantially increases the effective fetch/decode bandwidth during runahead mode, i.e., we are fetching/decoding multiple loop iterations at once, and (2) it requires very few back-end hardware resources, i.e., a vector instruction in vector-runahead mode corresponds to multiple scalar instructions from multiple loop iterations in the original code, while occupying only a single issue-queue slot.

The above example illustrates that Vector Runahead boosts runahead performance by issuing multiple loop iterations in parallel. We can achieve even higher performance by generating more MLP at an even faster rate through vector unrolling and pipelining, as we will describe in the next section. Vector unrolling issues multiple rounds of vector runahead (before returning to normal mode), which vector pipelining reorders so the processor can issue multiple vector copies of each instruction in immediate succession. For example, 8 rounds of vector runahead with 8 loop iterations per round enables speculating across 64 loop iterations in parallel.

Note that Vector Runahead does not require the original code to be vectorizable, nor does it require verifying that the vectorized instructions preserve the exact behavior of the original sequence — it only vectorizes the instruction stream during runahead mode to increase the performance of memory-latency-bound code by allowing many cache misses to become eligible for prefetching at once, increasing load coverage upon our return to normal execution. By doing so, Vector Runahead successfully prefetches indirect memory accesses, reducing the time the processor stalls on indirect memory accesses to 19.9% on average, see Figure 1. The next section describes the mechanics of Vector Runahead in detail.

III. Vector Runahead

Implementing Vector Runahead requires several modifications to the core microarchitecture, which we now describe.

A. Microarchitecture Overview

Figure 3 shows our baseline OoO pipeline with the modified and newly added hardware structures to support Vector Runahead. The stride detector [19] (Section III-B) is used to find regular access patterns in the code that can be used as 'induction variables' to produce speculative vectorized copies of code. Once we enter *vector-runahead mode* (Section III-C), instructions dependent on this vectorized stride pattern are tracked by a *taint vector* (Section III-D), and vectorized themselves (Section III-E): address-calculating arithmetic operations are converted into vector-unit operations, and the dependent loads themselves into vector gathers. Branches are assumed to match across each vectorized copy, with masking used to handle other cases (Section III-F). To further improve memory-level parallelism to greater levels than supported by a single vector load, we design *vector unrolling and pipelining* techniques (Section III-G) to issue many future loads simultaneously. Since this results in a one-to-many relationship between old scalar instructions and new vector instructions, a *vector register allocation table* (VRAT) is introduced at the front-end for register allocation (Section III-H), and a *register deallocation queue* (RDQ) at the back-end (Section III-I). Vector-runahead mode is terminated (Section III-J) once we have completed each unrolled iteration, where each iteration executes all loads dependent on the vectorized stride load.

B. Detecting Striding Loads

To detect sequences in the code from which we can generate induction variables to vectorize future memory accesses, we use a simple reference prediction table [19, 62] that is updated after the execution of each load instruction. This is indexed by the load PC, and each entry maintains four fields: (1) the last accessed memory address; (2) the last observed stride for the load; (3) a 2-bit saturating counter to indicate *confidence*; and (4) the *terminator* or the PC of the final dependent load in the instruction chain from the strided load.

From this, once we enter vector-runahead mode, we can generate streams of new instructions, to generate future indirect memory accesses based on the strided load, computation, and other intermediate memory accesses. The first three fields, (1) through (3), are standard for reference prediction tables [19]. The final field (4) is new and filled in during a round of vector runahead, and allows us to terminate early once all useful work finishes in vector-runahead mode (Section III-J).

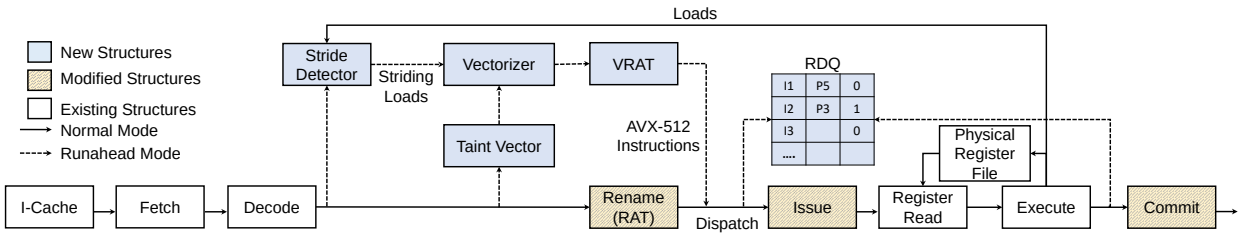


Fig. 3: Processor pipeline for vector runahead execution.

C. Entering Vector Runahead

The core enters *runahead mode* when either of the following two conditions is satisfied after a load instruction blocks the head of the ROB: (1) the ROB is filled with instructions; or (2) the issue queue is filled to 80% of its full capacity. Vector Runahead checkpoints the PC and the front-end register allocation table (RAT) by storing one checkpoint per entry of the front-end RAT, in addition to the checkpoints stored for recovery from branch misprediction. This marks the *entry* to runahead mode. The processor state will be restored to this checkpoint when we return to *normal mode*.

After entering *runahead mode*, the processor continues to fetch, decode, and execute future instructions. We access the stride detector for each load instruction. Until we reach a strided load, or if no such striding load exists, Vector Runahead performs similarly to PRE [64], though without the use of its fully-associative stalling-slice table, which Vector Runahead eliminates the need for, to avoid harming workloads without such patterns, and to capture any scalar dependencies later used by Vector Runahead. Equivalently, this mode behaves like a traditional runahead execution [57], only with active register reclamation and efficient checkpointing [64].

Entry to *vector-runahead mode* begins when we decode a striding load (with *confidence* = 3). We vectorize the striding load, followed by the sequence of instructions depending on it. The process terminates when another dynamic instance of the same striding load is detected, or the dependent chain is complete (Section III-J). We call the dependent instructions between two dynamic striding-load instances an *indirect chain*.

D. Taint Vector

To keep track of which operations (transitively) depend on the newly vectorized striding loads in the instruction stream, we use the *taint vector (TV)*. This features an entry for each architectural integer register, and stores two flags: (1) if the previous instruction to write to this register was a vectorized operation (*vectorize bit*); and (2) if the previous instruction to write to this register was invalid (*invalid bit*). The TV is empty at the start of runahead, as it is cleared whenever runahead terminates. Vectorize bits are initially set for the destination architectural register of a discovered striding load. Invalid bits are initially set based on the destinations of unsupported operations, e.g., those that take floating-point operations as input (which are always invalid and so need no TV entry). Both bits are propagated using *vector taint tracking*, a mechanism to propagate vectorization where needed. If any of an instruction's input registers are tagged, then the destination register becomes tagged as well. If no input registers are tagged, the destination register's flag is unset. Instructions with no bits set are issued as conventional scalar runahead operations, and treated as loop-invariant with respect to vectorized copies of the instruction sequence in the current vector-runahead mode iteration. Instructions with the invalid bit set are discarded, and instructions with only the vectorize bit set are vectorized.

E. Vectorizing Instructions

Vectorization is performed via a microprogrammed routine that generates vectorized versions of input scalar instructions. For striding loads, the vectorizer generates their vectorized versions by taking the current memory address accessed by the striding load and its stride as inputs. The vectorizer generates one 512-bit vector load instruction and *injects* the vector instruction into the pipeline (see Section III-G for generating multiple 512-bit vector instructions). Regardless of input bit width, eight scalar operands are fit in this 512-bit vector, such that we can operate on any size up to 64 bits. We assume that each vector instruction uses 512-bit vector registers (similar to Intel AVX-512) for its source and destination, and we reuse the microarchitecture's physical vector registers, and the micro-ops implemented by the microarchitecture's vector units.

Similarly, we vectorize all arithmetic and load instructions (directly or indirectly) depending on a striding load, and generate their corresponding 512-bit vector versions. All vectorized instructions are renamed using the vector register allocation table (VRAT) (Section III-H). The renamed instructions are dispatched to the processor back-end where they are executed speculatively. It is possible that there is a chain of load instructions depending on a striding load, for example, in the case of multiple levels of indirect dependent loads (as in pointer-chasing code). In such cases, all the load instructions forming the dependency chain are vectorized into gather operations, and the stride table's *terminator* (Section III-B) is updated, if empty at the start of runahead, with the PC of the latest dependent gather load. Therefore, Vector Runahead can generate memory-level parallelism for multiple levels of indirect memory accesses. If individual lanes within the vector generate invalid memory accesses, the individual lane is marked invalid, which causes lanes within subsequent vectorized instructions to be masked, and their execution ignored.

The instructions executed in runahead mode are useful only in generating memory accesses and their state is not maintained in the ROB. Therefore, no ROB entries are allocated in runahead mode. Instead, we use the simpler register deallocation queue [64] (RDQ, Section III-I) to handle register

availability. Since floating-point instructions are rarely used to calculate addresses themselves, we ignore such instructions (marking them and any instructions using them as invalid [57]), along with stores, and any instructions that are already vectorized in the original code.

F. Control Flow

When vectorizing, we make the implicit assumption that all vector lanes will follow the same pattern of control flow as each other. However, when executing in vector-runahead mode there may be divergence between the lanes when they meet a branch instruction. We use a micro-op that converts scalar branches into a predicate mask for the eight vector lanes. Since Vector Runahead need not cover all code, we then use only the results of the first lane to determine the direction of the branch, and mask off any lanes that would have taken a different control path. This masking persists until we terminate a single iteration of vector-runahead. By contrast, different *unrolled* iterations (Section III-G) within a single vector-runahead interval may follow independent control flow.

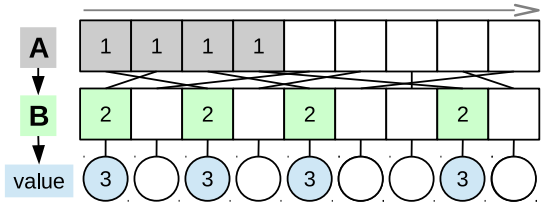
G. Vector Unrolling and Pipelining

This basic Vector Runahead suffers from the following key drawbacks, as illustrated in Figure 4(a) assuming a processor vector width of four[1]: (1) it does not move far enough ahead in the execution stream — limiting timeliness; (2) it does not spend enough time in runahead mode — limiting coverage; and (3) it does not issue enough simultaneous loads to saturate the miss status holding registers (MSHRs) — limiting MLP.

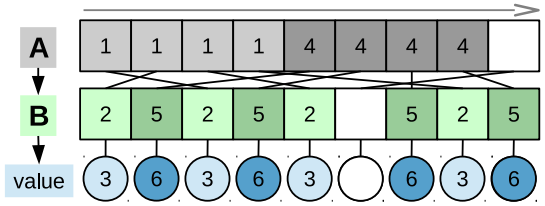
To solve the first two problems, we issue multiple rounds of vector runahead before returning to normal mode, in a process called *vector unrolling* (Figure 4(b)). Once the first round is complete, we issue a vector load for the next N values identified in the strided sequence, where N is the number of words in the vector (4 in this example or 8 for AVX-512). We then repeat this, incrementing the addresses of the strided loads, until we have issued U, the *unroll length*, copies of the vector-runahead sequence. In case $N = 8$ and $U = 8$, we issue 64 iterations of the original scalar loop before leaving vector-runahead mode.

Vector unrolling by itself does not solve the limited MLP problem. Even with AVX-512, we can only fit eight loads into a single gather, limiting the number of parallel memory accesses to eight before stalling on dependent loads. With smaller vector sizes, this problem becomes (even) worse. We hence introduce *vector pipelining*, a software-pipelining-style optimization [7, 41, 80] that reorders loads to issue multiple independent memory accesses in parallel. Instead of waiting for a previous round of Vector Runahead to finish before starting the next, we issue multiple copies of every vectorized instruction simultaneously, all with stride-load inputs at different lookahead distances, creating a one-to-many mapping between scalars and vector instructions. In Figure 4(c), we have an *unroll length* U of 2, and a *pipeline depth* P of 2.

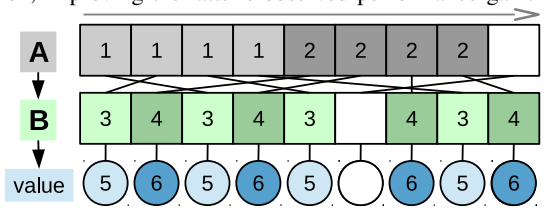
[1]A vector width of 4 is chosen in this example for illustrative purposes only. We consider a vector width of 8 in our simulated configuration.



(a) Basic Vector Runahead. In this example, MLP is limited to a single vector instruction, so only four outstanding memory accesses can be prefetched at once, and few future memory accesses are covered by the memory-parallel vector runahead, limiting performance gains for future normal execution.



(b) Vector Unrolling. While the Vector Runahead operations are still run in sequence, with a maximum MLP of 4, we cover significantly more of the future memory accesses before returning to normal execution, improving the latter's observed performance gain.



(c) Vector Pipelining. We overlap the independent operations from multiple unrolled iterations. This allows many misses to be handled simultaneously: in this example, 1 and 2 can be executed in parallel, doubling MLP to 8, as can 3 and 4, and 5 and 6.

Fig. 4: Vector Runahead uses two techniques, vector unrolling and vector pipelining, to improve performance by increasing the degree of runahead to allow wider vectors than supported natively by the instruction-set architecture.

This allows us to issue the two vector loads to A (1 and 2) in parallel, before moving on to their dependents (3 and 4) which are also issued in parallel, and then finally 5 and 6.

Though pipelining better allows us to extract memory-level parallelism, there are limits to this, as we will never be able to service more simultaneous cache misses than there are MSHRs. Unrolled vectors in distinct pipeline groups can reuse the same physical registers, as they are not live simultaneously. By contrast, vector pipelining deliberately overlaps the live range of physical registers for different unrolled iterations, and also increases the VRAT size (Section III-H). By default, we assume both an unroll length U and a pipeline depth P of 8, which allows us to issue vector gathers for 64 scalar loads simultaneously. While we hard-code these in our implementation, they could be tuned dynamically based on observed system performance, accuracy and register availability.

H. Vector RAT

For each scalar architectural register input, when the *vectorize bit* is set, we must redirect the new vector instruction

to the appropriate source vector physical register. Normally, the register allocation table (RAT) renames architectural scalar registers into physical scalar registers. However, we must rename into physical vector registers instead, and with the addition of vector pipelining (Section III-G), we have a one-to-many relationship between architectural scalar registers and renamed physical vector registers. For a pipeline depth of 4, we need to rename an architectural scalar register into 4 separate vector physical registers (each covering 8 data elements). This means we add a new *vector register allocation table (VRAT)*, with P entries per architectural integer register, recording the P destination physical vector registers assigned to the P pipelined copies of the instruction. When we look up these P registers in the VRAT, each of the P copies of the new vectorized instruction use one of the P entries as its own input. This enables us to distinguish the inputs and outputs of separate pipelined iterations within the vector-pipelining arrangement, which, from an instruction fetch point of view, all alias to the same instruction. We need P entries for each of the 16 integer registers, which is typically small in size.

I. Managing Pipeline Resources During Runahead

There must be a sufficient number of unused issue queue and physical (scalar and vector) register file entries for speculatively executing indirect chains leading to indirect loads. In Vector Runahead, one vector instruction occupies one issue-queue entry from dispatch to execution. Upon execution, the issue-queue entry is freed and can be allocated to a younger instruction, similar to a standard OoO core.

Unlike the issue queue, physical register file entries cannot be released out-of-order. In an OoO core, a physical register is freed only when a new instruction writing to the same architectural register to which the physical register was mapped, is committed [78]. Since we do not commit instructions in runahead mode, the core may stall due to unavailability of physical registers, so Vector Runahead frees a physical register when the register is no longer required for address generation. This is done via a simple in-order *register deallocation queue (RDQ)*, also used by PRE [64]. Each instruction is looked up in the VRAT to see the P (under vector pipelining) physical vector registers holding the destinations of the instruction that last wrote to the same architectural register, which will become dead once the new instruction reaches the end of the pipeline. P *vector-pipelined* instruction copies, generated from a single scalar instruction and issued as independent vector operations, each have their own RDQ entry, covering one of the P soon-to-be-dead physical vector registers.

Figure 5 shows the RDQ operating on the hot loop from the RandAcc benchmark, assuming $P = 2$. Instruction #1, and its pipelined copy #2, are striding loads, which load addresses based on the stride detector. Before we enter vector-runahead mode, we assume that $S1..3$ are scalar physical registers that currently store `rdx`, `rdi` and `rbp`, respectively; $S4$ is a free physical scalar register; and $P1..15$ are free physical vector registers. All instructions except #3, and the invalid stores, form two indirect chains leading to the indirect loads #16 and #17, so all apart from instruction #3 and the stores are vectorized. Next to each instruction we show the source and destination physical register IDs and the contents of the RDQ, which consists of the physical register ID to free (when safe to do so) and a bit to indicate whether the instruction has been executed. An RDQ entry is allocated for each vectorized instruction; no RDQ entry is allocated for invalid instructions such as stores — marked in dark gray. The RDQ maintains a head pointer that points to the first unexecuted instruction; in our example, the head pointer points to instruction #7 — all younger instructions have been executed and are marked in light gray. When an instruction reaches the head of the RDQ and has been executed, the RDQ frees the physical register indicated and moves the head pointer on. The only vector register to have been released so far is $P1$, which is reused for instruction #17. Once #7 executes, registers $P2$ and $P5$ will be released (since instruction #8 has also been executed), and the head pointer will move to instruction #9.

Renaming AVX Registers in Vector Runahead

inst. id	instruction	dst	src1	src2
1	mov rcx, qword ptr [rdx]	P1	S1	
2	*Pipe Inst. #1*	P2		
3	add rdx, 0x8	S4	S1	
4	lea rax, ptr [rcx+rcx*1]	P3	P1	
5	*Pipe Inst. #4*	P4	P2	
6	sar rcx, 0x3f	P5	P1	
7	*Pipe Inst. #6*	P6	P2	
8	and ecx, 0x7	P7	P5	
9	*Pipe Inst. #8*	P8	P6	
10	xor rax, rcx	P9	P3	P7
11	*Pipe Inst. #10*	P10	P4	P8
12	mov rcx, rax	P11	P9	
13	*Pipe Inst. #12*	P12	P10	
INV	mov qword ptr [rdx-0x8], rax	INV	S4	P9
INV	*Pipe (INV)*	INV	S4	P10
14	and rcx, rdi	P13	P11	S2
15	*Pipe Inst. #14*	P14	P12	S2
16	xor qword ptr [rbp+rcx*8], rax	P15	P13	S3
17	*Pipe Inst. #16*	P1	P14	S3

Register Deallocation Queue

inst. id	register to free	executed ?
1		1
2		1
3		1
4		1
5		1
6	P1	1
7	P2	0
8	P5	1
9	P6	0
10	P3	1
11	P4	0
12	P7	0
13	P8	0
14	P11	0
15	P12	0
16	P13	0
17	P14	0

Fig. 5: Example of an instruction sequence in vector-runahead mode, with duplication of vectorized instructions via vector pipelining $P = 2$, and associated renaming. The RDQ is filled in as a result of lookup in the VRAT, which can be used to reuse destination registers once no longer live.

J. Terminating Runahead

Vector-runahead mode terminates when any of the following four conditions is satisfied: (1) we encounter a dynamic instance of the same striding load again; (2) we encounter, and issue, the *terminator*: the PC identified by the stride detector (Section III-B) as the last dependent load in the sequence; (3) all vector lanes have been marked as invalid; or (4) we time out (after 200 scalar-equivalent instructions have been executed in vector-runahead mode), in case of traveling down an unexpected code path. When $U > P$ (Section III-G), i.e., the unroll length is greater than the pipeline depth, we re-enter vector-runahead mode immediately, with the next striding load

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

(a) C code

inst. id	instruction
1	mov edx, dword ptr [rax]
2	add rax, 0x4
3	mov ecx, edx
4	shr ecx, 0x10
5	xor ecx, edx
6	imul edx, ecx, 0x45d9f3b
7	mov ecx, edx
8	shr ecx, 0x10
9	xor ecx, edx
10	imul edx, ecx, 0x45d9f3b
11	mov ecx, edx
12	shr ecx, 0x10
13	xor ecx, edx
14	and ecx, 0x1fffff

inst. id	instruction
15	mov edx, dword ptr [rcx*4+0x18602220]
16	mov ecx, edx
17	shr ecx, 0x10
18	xor ecx, edx
19	imul edx, ecx, 0x45d9f3b
20	mov ecx, edx
21	shr ecx, 0x10
22	xor ecx, edx
23	imul edx, ecx, 0x45d9f3b
24	mov ecx, edx
25	shr ecx, 0x10
26	xor ecx, edx
27	and ecx, 0x1fffff
28	add dword ptr [rcx*4+0x106021e0], 0x1

(b) Assembly

Fig. 6: Example hot loop from Kangaroo with a striding load (#1) followed by two dependent indirect loads (#15 and #28).

issuing vector gathers again. This is repeated until we have issued U/P total rounds and only then is normal execution resumed. As we show in Section VI, the benefit of vectorizing the entire indirect chain far exceeds the additional duration the core is in runahead mode, as Vector Runahead yields higher memory-level parallelism than typical out-of-order execution.

Upon termination, we restore the front-end RAT to the point of entry into runahead mode, and the TV, VRAT and RDQ are cleared. The front-end is redirected to fetch from the next instruction after the last dispatched instruction in the ROB.

K. Hardware Overhead

Vector Runahead incurs few new structures, and they are all small; the other Vector Runahead structures repurpose existing components. The stride detector (48-bit last address, 16-bit stride distance, 2-bit counter, 48-bit terminator, 32 entries) requires 456 bytes of storage. The taint vector uses 4 bytes of storage (2 bits for each of 16 registers), and the VRAT incurs 112 bytes (to encode a mapping to eight pipelined physical vector registers for each scalar architectural register). The RDQ, with 192 entries as used by PRE [64], takes up 768 bytes. In all, the SRAM overhead for Vector Runahead is just 1.31 KB, compared with PRE at 1.24 KB [64].

IV. Representative Code Example

We now contrast Vector Runahead against the state-of-the-art Precise Runahead Execution (PRE) [64] through a representative code example from the Kangaroo benchmark, see Figure 6. This hot loop consists of one striding load (#1), which fetches a value that is hashed to generate the indirect address for the first indirect load #15, which in turn is hashed to generate the address for the second indirect load (#28). Note that all instructions from #3 to #14 are required to generate the indirect addresses for #15; similarly, #15 and all instructions from #16 to #27 are required to generate the addresses for #28; finally, #2 is needed for #1 in the next loop iteration.

The behavior that we observe for PRE is very similar to what we described in Figure 2(c). Since our baseline procesor includes a stride prefetcher, the #1 loads typically hit in the L1 D-cache, enabling the quick issuing of the #15 loads. The indirect nature of #15 causes a DRAM access and delays the execution of any #28 load. In other words, no copies of #28 will be executed. Furthermore, the degree of MLP that can be exploited is limited by the number of #15 loads that make it into the issue queue. Because instructions #16 to #28 occupy issue queue slots, the exploitable degree of MLP is limited by how many copies of instructions #16 to #28 can fit in the issue queue during runahead mode. We find that MLP saturates around 5 for PRE, as we will later quantify in the evaluation section (see Figure 8 in particular).

In contrast, Vector Runahead generates a substantially higher degree of MLP. After entering into runahead mode, Vector Runahead initiates vectorization when it hits a dynamic instance of the striding load #1. Assuming $U = 8$ and $P = 8$, Vector Runahead generates eight vector instructions for #1, and continues the vectorization until it hits #28; note that this corresponds to 64 iterations of the original loop. In contrast to PRE, Vector Runahead's ability to issue 64 independent scalar-equivalent loads in immediate succession allows it to generate substantially higher MLP. The processor remains in vector-runahead mode until all eight vectorized copies of #28 are issued, to maximize coverage for all loads (including #28) when the processor resumes to normal mode.

Note that in addition to higher degrees of MLP, Vector Runahead also generates the MLP at a much faster pace compared to PRE. Whereas PRE is fundamentally limited by the processor's fetch/decode bandwidth, and how many operations can fit within the issue queue, Vector Runahead generates many more independent memory accesses in immediate succession. The effect is that Vector Runahead dramatically increases the effective fetch/decode bandwidth during runahead mode. In other words, Vector Runahead generates more MLP at a much faster rate.

V. Experimental Setup

A. Simulation Setup

We use the most accurate superscalar core model in Sniper 6.0 [15], a cycle-level and hardware-validated simulator, which we adjust to faithfully model Vector Runahead. Table I provides the configuration of our baseline out-of-order processor based on Intel Skylake [24]. We consider an aggressive PC-based L1-D stride prefetcher supporting 16 streams, and 24 MSHRs to keep track of outstanding cache misses. The branch predictor is an 8 KB TAGE-SC-L from the 2016 Branch Prediction Championship [75]. We skip initialization and run for 200 million instructions within the workloads' representative regions of interest.

B. Workloads

We consider a variety of benchmarks featuring complex memory and compute dependencies in their execution stream. These benchmarks are memory latency bound on today's systems, and are based on high-performance computing (HPC),

Core	3.2 GHz, out-of-order
ROB size	224
Queue sizes	issue (97), load (64), store (60)
Processor width	4-wide fetch/dispatch/rename, 8-wide commit
Pipeline depth	8 front-end stages
Branch predictor	8 KB TAGE-SC-L
Functional units	3 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)
Register file	180 int (64 bit) 180 fp (128 bit) 96 vector (512 bit)
L1 I-cache	32 KB, assoc 4, 2-cycle access
L1 D-cache	32 KB, assoc 8, 4-cycle access, 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	45 ns min. latency, 51.2 GB/s bandwidth, request-based contention model

TABLE I: Baseline configuration for the out-of-order core.

graph and database workloads evaluated in previous work on programmer- and compiler-managed prefetching mechanisms [3, 4]. The benchmarks' basic parameters of interest are provided in Table II. The benchmarks are:

- Camel [4] (2 hashes input), a workload with significant computation between indirect memory accesses;
- Graph 500 [56] with inputs `-s 16 -e 10` and `-s21 -e 10`, a graph breadth-first search;
- Hash Join [13] (2 elements per bucket and 8 elements per bucket [3]), a database kernel where a hashing calculation must be performed to generate chains of addresses;
- Kangaroo [4] (3 arrays, 2 hashes), a workload with long indirect chains and complex address computation;
- Conjugate Gradient (CG) and Integer Sort (IS) from the NAS Parallel Benchmark Suite [11] (input B), supercomputing kernels with indirect memory accesses where CG's indirect memory access is likely to fit in the last-level cache (LLC); and
- RandomAccess [50], an HPC kernel with complex indirect address computation.

These benchmarks represent a variety of different complex memory-access patterns, with differing indirect chains and compute requirements. We use compiler flag *-ftree-vectorize* (via O3) in all comparisons, but we find that autovectorization does not alter performance because the code is not vectorizable (despite being amenable to Vector Runahead).

VI. Evaluation

We compare the following microarchitectural mechanisms:

- Out-of-Order (OoO): Baseline out-of-order core as shown in Table I, with hardware stride prefetcher.
- Precise Runahead Execution (PRE): The state-of-the-art runahead execution technique, as proposed by Naithani et al. [64]. We assume an ideal stalling-slice table; therefore, there are no misses in the table.
- Indirect Memory Prefetcher (IMP): The indirect memory prefetcher, as proposed by Yu et al. [88]. IMP is attached to the L1 D-cache; it detects indirect access patterns starting from striding memory accesses.
- Vector Runahead (VR): The mechanism proposed in this paper. Our representative vector runahead technique, with *unroll length* U of 8 and *pipeline depth* P of 8, unless mentioned otherwise.

Workload	*Ind. Chain*	*Arithmetic*	*Insts per Load*
Camel	2	×	15
Graph500-s16	2–4	×	7
Graph500-s21	2–4	×	5
HJ2	2	✓	7
HJ8	5	✓	6
Kangaroo	3	✓	11
NAS-CG	2	×	4
NAS-IS	2	×	4
RandAcc	2	✓	8

TABLE II: A summary of our workloads, including the length of the indirect chain (including striding load) within the most common inner loops; whether complex arithmetic (beyond basic array indexing) is necessary to calculate the addresses needed for prefetching in runahead mode; and how many total instructions occur on average per load at run-time.

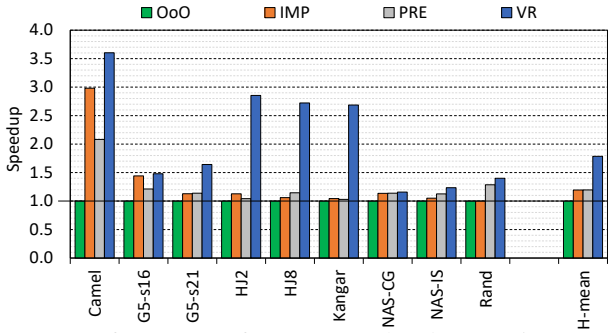


Fig. 7: Performance of vector-runahead execution on the baseline out-of-order core. *Vector Runahead yields a 1.79× and 1.49× harmonic mean speedup compared to the baseline OoO core and PRE, respectively.*

A. Overall Performance Comparison

Figure 7 reports speedup for all the evaluated techniques. Vector Runahead achieves a 1.79× harmonic mean speedup across the benchmarks compared to our baseline OoO architecture. The achieved speedup is as high as 3.6× (Camel), 2.9× (HJ2), 2.7× (HJ8) and 2.7× (Kangaroo). PRE on the other hand achieves a harmonic mean speedup of 1.20× compared to the baseline — in other words, Vector Runahead achieves a speedup of 1.49× relative to PRE. IMP cannot detect complex address computation patterns and improves speedup by only 1.19× relative to the baseline. In short, the significant improvement in performance achieved by Vector Runahead results from much higher memory-level parallelism (Figure 8), while fetching in all loads within dependent sequences (Figure 12), and without fetching irrelevant data (Figure 13(a)), as we elaborate in the next few subsections. Before doing so, we first discuss where PRE falls short.

PRE improves performance by processing future instructions beyond the ROB after a full-window stall. The perfor-

mance improvement through PRE depends on the number of indirect accesses covered in each runahead interval. Since the baseline system has a stride prefetcher attached to it, the *stride* part of a single-level stride-indirect (*indirect chain* of 2) may be in the cache for runahead execution to then use, allowing it to prefetch the single indirect memory access somewhat successfully (e.g., Camel and CG). Still, this is ineffective for any workload with an *indirect chain* greater than two, where the lack of access to intermediate loads for address calculation hinders coverage, limiting the amount of MLP exposed by PRE over standard out-of-order execution. This holds true even if we modify PRE to stay in runahead mode as long as Vector Runahead does to prefetch indirect chains, where the extra performance improvement is limited to 3.5% on average.

IMP is also able to improve performance for applications with primarily single-level indirect patterns (e.g., Camel and CG). However, for more complex chains of instructions between the striding and indirect loads (e.g., HJ8, Kangaroo and RandAcc), IMP cannot perform the necessary computation.

Vector Runahead achieves higher performance by three main mechanisms. The most important is the software-pipelining effect that reordering of load instructions gives us, in that a large number of misses can be serviced simultaneously. This same reordering, implemented with 64 scalar micro-ops instead of 8 vector micro-ops, is still sufficient to gain an average 1.47× speedup. The optimization of packing these into fewer vector operations, due to their now-SIMD layout, increases performance to 1.69× by virtue of improving compute throughput and by requiring fewer issue queue slots so that loads can issue earlier. Finally, altering the termination condition, such that Vector Runahead completes the entire chain of memory accesses before exiting, allows it to cover longer chains, of multiple main memory accesses, rather than just the ones it can achieve before the head of the ROB returns, increasing performance to the full 1.79× shown in the graph.

B. Performance and Sensitivity Analysis

Memory-Level Parallelism. Figure 8 shows why Vector Runahead is able to achieve higher performance. Its pipelined vectors are able to issue many gathers to memory at once, thus hiding the serialization of dependent loads observed by the out-of-order core and PRE. This also shows us why some workloads are sped up more than others. Although our baseline OoO core features a relatively big ROB, which enables it to achieve high MLP on the simplest workloads, we note that Vector Runahead can extract significantly more MLP. Perhaps unsurprisingly, Vector Runahead achieves the largest speedups when the out-of-order core is comparatively weakest: for Camel, HJ2, HJ8 and Kangaroo, there are many instructions (address-computing or otherwise) executing along with the loads (Table II), which starve the out-of-order core of reorder buffer and issue queue resources [4, 5], limiting its memory-reordering ability. By contrast, Vector Runahead does not rely on the reorder buffer for high memory-level parallelism, as it can achieve the same effect through its vector gathers.

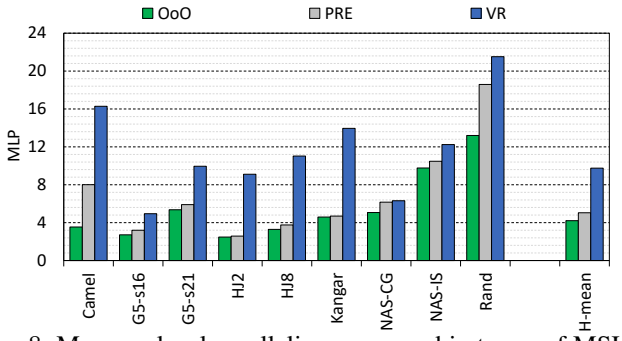


Fig. 8: Memory-level parallelism measured in terms of MSHR entries utilized per cycle if at least one is allocated. *While precise runahead improves MLP by 1.2×, vectorizing indirect chains generates 2.3× more MLP than an OoO core.*

Some workloads, such as G5-s16 and G5-s21, start from a low baseline and stay relatively low even with Vector Runahead: complex control flow limits the ability of Vector Runahead to cover enough of the application's memory accesses, in effect throttling the vector gathers issued, particularly for the smaller s16 input, which frequently moves between variable-length data-dependent inner- and outer-loops. Others, such as CG and G5-s16, have small datasets that often hit in the LLC, meaning their L1 cache misses are serviced quickly with or without Vector Runahead. Finally, even though many workloads end up MSHR-constrained within vector-runahead mode (Figure 11), thus achieving the highest MLP the CPU can feasibly achieve, the average MLP is still lower than 24, the maximum achievable with 24 MSHRs: this is because Vector Runahead cannot run continuously, and only kicks in when the out-of-order system runs out of resources.

Unrolling and Pipelining Analysis. Our default setting of an unroll length U of 8 and a pipeline depth P of 8, allows us to issue 64 scalar-equivalent copies of a dependent load chain at a time. We see in Figure 9 that with no unrolling or pipelining ($U = 1, P = 1$), since we only issue 8 scalar loads at a time (and in total) before leaving runahead, almost no performance improvement is registered; the limit on MLP of the eight loads in a single gather means we struggle to beat the OoO core's performance, and so the compute time we take away from it to execute in runahead is barely worth the effort. Issuing larger unroll lengths without increasing the pipeline depth (e.g., $U = 8, P = 1$) improves performance slightly, by increasing coverage of Vector Runahead, but still lacks in memory-level parallelism. Still, $U = 8, P = 4$ (32 loads at once) is often sufficient to saturate 24 MSHRs; $U = 8, P = 8$ gains only a small amount more performance overall.

LLC Size. As we see in Figure 10, a larger cache is no substitute for Vector Runahead. The workloads it targets feature large datasets, which is typical of today's big-data workloads, that do not fit in any reasonably sized cache, meaning neither the baseline nor the performance of Vector Runahead is affected significantly by large cache sizes. The speedup obtained through Vector Runahead is largely invariant.

Number of MSHRs. Figure 11 shows the IPC, relative to our

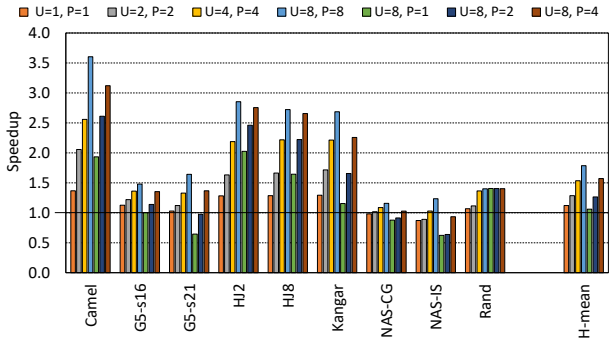


Fig. 9: Performance impact of unrolling and pipelining. U (unroll length) is the total number of *separate copies* of the vectorized scalar instructions. P (pipeline depth) is the number of vector instructions launched simultaneously in each round.

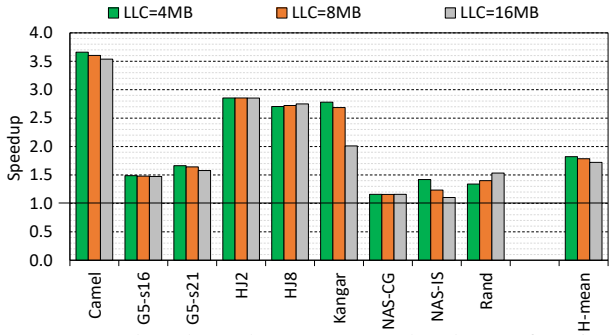


Fig. 10: Speedup through Vector Runahead as a function of LLC size. Performance is normalized to a baseline OoO core with the same respective LLC size. *Increasing LLC size has minor impact on performance due to the indirect memory accesses in these workloads. Hence, Vector Runahead's speedup is largely invariant.*

baseline 24-MSHR setup, from varying the number MSHRs in the cache. As Figure 8 previously demonstrated, 12 MSHRs is insufficient to achieve the full benefits of Vector Runahead; contention between concurrent misses for MSHRs limits the achievable MLP. Still, even though our average utilization in Figure 8 is always below 24 MSHRs, 36 and 48 still gain small improvements for a few workloads: during vector-runahead mode with the vector-pipelining $P = 8$ setting, we can theoretically issue 64 gather loads simultaneously, and some workloads issue more of these in parallel, thus leaving vector-runahead mode more quickly, with more MSHRs.

C. Vector Runahead Effectiveness

Coverage. Figure 12 shows the number of off-chip memory accesses issued by the core under Vector Runahead versus PRE, relative to our baseline. It also shows the fraction of these that are issued in normal mode only, and thus have been adequately prefetched during runahead mode. There are two key observations. First, neither technique significantly overfetches: both generate few unnecessary memory accesses in their pursuit of MLP. Second, Vector Runahead has much higher coverage: the proportion of memory accesses during normal mode is small for Vector Runahead (11.5% on average)

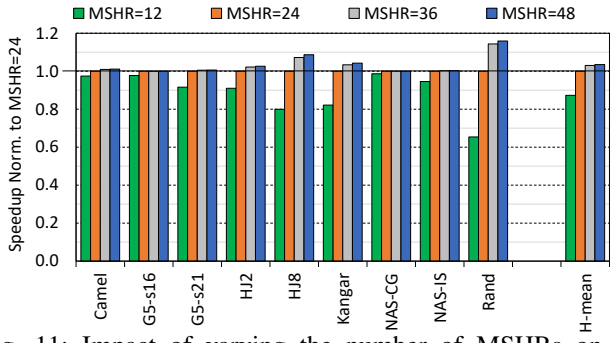


Fig. 11: Impact of varying the number of MSHRs on the performance of Vector Runahead. *A sufficient number of MSHRs is needed to fully benefit from Vector Runahead.*

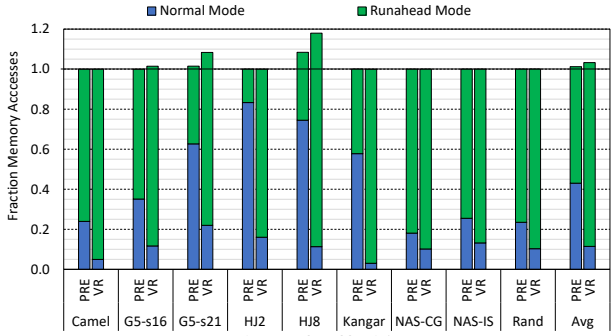


Fig. 12: Coverage: number of off-chip memory accesses for PRE and Vector Runahead normalized to OoO. The graph also shows the fraction of memory accesses in normal versus runahead mode. *Vector Runahead successfully prefetches DRAM accesses, converting them into on-chip cache hits when the program actually accesses them during normal mode.*

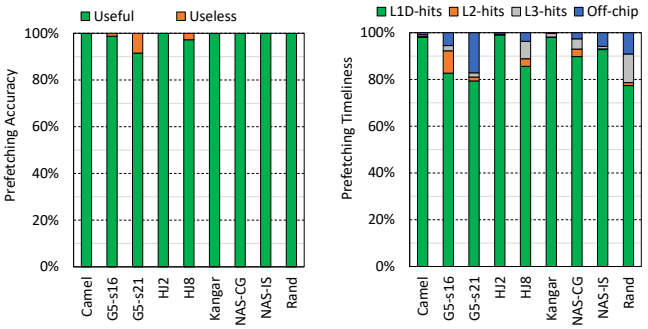
compared to PRE (43%), due to its comprehensive handling of indirect chains.

Accuracy. The low number of excess DRAM reads is echoed in Figure 13(a). Vector Runahead rarely brings in data during runahead mode that is not later used in normal mode before it leaves the cache hierarchy. The exception is when the dataset is large and the control flow complex, causing the stride detector to generate vector addresses that are not later accessed, as is the case for Graph500-s21, but still accuracy is over 90%, and net performance improvement is high.

Timeliness. Despite the high MLP exploited by Vector Runahead, its prefetches are timely: we see in Figure 13(b) that most are in the L1 cache by the time they are used, and while some have been evicted into the L2 or L3 caches, these are still preferable over high-latency off-chip accesses. Since Vector Runahead can terminate as soon as it issues the last load in a chain, some prefetches are still incomplete by the time normal mode accesses them, and so a portion of the *off-chip* memory latency is still observed through MSHR hits during normal mode. Overall, most are ready by the time normal execution reaches them.

VII. Related Work

Vector Runahead is a form of prefetching, summaries of which have been performed by Mittal [53], and Falsafi and



(a) Accuracy (b) Timeliness

Fig. 13: (a) Accuracy: fraction of total prefetched cachelines in runahead mode that are later accessed in normal mode. (b) 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 the cachelines for which the data is still being transferred from memory. *Vector Runahead rarely brings in data during runahead mode that is later not used during normal mode, and most cachelines prefetched during runahead mode turn into short-latency cache hits during normal mode.*

Wenisch [28]. The most relevant work is categorized below.

A. Auto-vectorization

Vector Runahead has its roots in auto-vectorization [51, 69, 70] and software pipelining [18, 41, 73, 80, 81] to improve the number of load misses serviced simultaneously under limited front-end and back-end resources. In contrast to these static software techniques, Vector Runahead is a dynamic hardware technique, which does not need to be correct to adequately prefetch into the cache.

B. Runahead Execution

Runahead execution [25, 57] is a microarchitecture technique designed to skip ahead of long-latency loads: runahead execution unblocks the processor when blocked by a long-latency load to prefetch independent memory accesses long before they are needed. Mutlu et al. throttle [58, 60] and redesign [61] for increased efficiency. Filtered runahead [32] uses backwards dataflow to directly target instruction sequences. Precise runahead execution [64] further improves performance with more efficient checkpointing techniques, register reclamation, and filtering of unnecessary instructions. Continuous runahead [34] offloads execution to an accelerator, to avoid sharing resources with the main core; while the in-order core would not directly be able to achieve the high memory-level parallelism needed for indirect workloads, Vector Runahead also could hypothetically be offloaded to such a processor. Load Slice Core [16], Freeway [43] and Forward Slice Core [46] slice up programs, offloading memory accesses to in-order queues; Vector Runahead could further increase the exploitable MLP in these slice-out-of-order cores. Other work improves runahead's efficiency [71, 72, 82, 83].

Runahead execution's inability to target dependent memory accesses is well known in the literature. Address-value delta prediction [59, 62] uses a value predictor to predict address-generating load values, instead of directly performing the loads themselves in a vector style, as Vector Runahead does. For sequential list walks without memory-level parallelism, Hashemi et al. [33] move calculation into the memory controller.

Fundamentally, Vector Runahead presents a solution for achieving memory-level parallelism down dependent chains, which gives the above mechanisms the ability to deal with complex memory-access patterns. While our implementation considers this runahead within-thread and within-core, as in the original runahead paper [57], the technique is more general, and could equally be applied to mechanisms that offload the runahead work [33, 34, 71, 72, 82, 83].

C. Pre-Execution and Helper Threads

More generally, techniques exist in the literature that precompute memory accesses. Dependence-graph computation [8, 74] uses a separate execution engine to precompute instruction sequences for memory accesses. Slice Processors [54] similarly extract instruction sequences to generate loads. Pre-execution of loads can also be done through other threads. Kim and Yeung [38] use the compiler to generate 'helper threads', which bring in data for the main computation thread. Speculative Precomputation [22] allows speculative helper threads to spawn their own speculative helper threads, to target complex chain dependencies, which Vector Runahead handles via vectorization. Lau et al. [47] offload these to small partner cores. Ultimately, both helper threading and runahead execution have roots in Decoupled Access Execute [77], where load and compute slices are executed independently. DeSC [31] splits up the entire processor into access- and execute-helper threads. Ganusov and Burtscher [29] emulate microarchitectural prefetchers via helper threading. By contrast to these techniques, Vector Runahead needs no separate thread, no separate execution units, and neither programmer nor compiler support. Moreover, since Vector Runahead can follow dependent chains, unlike pre-execution and helper threads, it can achieve substantially higher memory-level parallelism.

D. Architecturally Visible Prefetching

The workloads that Vector Runahead targets have mostly been targeted by prefetchers that require software support. Harbinger instructions [6] and Guided-Region Prefetching [84] introduce the idea of controlling a hardware prefetcher with programmer hints. Specialized configurable prefetchers have been developed for Graph workloads [1] and linked structures [21, 42]. RnR [89] uses programmer hints to assist temporal history prefetchers, discussed below. Cavus et al. [17] present an array-tracking prefetcher. More generally, programmability can be added within the memory hierarchy [87]. The Event-Triggered Programmable Prefetcher [3] extracts memory-level parallelism via extreme thread-level parallelism, by contrast to the data-level parallelism within a core we use for Vector Runahead, on a many-core accelerator. *Fetchers*, which control memory accesses directly, and thus change program semantics instead of being prefetch hints, include

Minnow [90], Meet the Walkers [40], SQRL [44], DASX [45], Ho et al. [35], Livia [48] and Pipette [66].

Software prefetching [14, 18] is a widely deployed example of a prefetching technique that requires software support, where special non-blocking loads are inserted into the program stream. Mowry [55] develops algorithms to insert software prefetches in the compiler. Ainsworth and Jones develop compiler techniques for indirect memory accesses [2, 4]. Software prefetching can also be used as a building block to enable fetcher-unit-like behavior: AMAC [41] uses software prefetching to emulate Walkers [40]. More generally, instructions can be software-pipelined in the compiler to better extract memory-level parallelism, such as in Clairvoyance [80]. By comparison, Vector Runahead is microarchitecture-only, requiring no changes to the binary or source code, and since it can speculate within runahead mode, Vector Runahead can freely vectorize sequences of instructions that would cause software prefetchers to fault.

E. Microarchitectural Prefetchers

Most prefetchers deployed to date require no software support, sitting in the microarchitecture. Stride prefetchers [19, 20], which pick up sequences in address patterns, have seen wide use in commercial systems [10], and a wide body of work exists to improve their coverage, performance and selectivity [12, 39, 52, 76]. Temporal-history prefetchers [36, 37, 65, 85, 86] store and replay histories of past cache misses, but cannot support the volume of data necessary for, or lack of repetition in, complex big-data workloads [1, 88]. Cooksey et al. [23] present a content-directed prefetching mechanism to fetch any data brought into the cache that appears to be a pointer, regardless of whether it is used by the program, though compiler input [6, 26] is typically necessary to throttle over-fetching. The Bouquet of Prefetchers [68] uses multiple different styles of prefetcher, selected by a PC-based predictor. IMP [88] targets stride-indirect memory access patterns in the cache, as do Takayashiki et al. [79] via observing gathers. In contrast, Vector Runahead operates within-core, allowing it to cover arbitrary indirection depths with complex address calculation, as needed in many workloads [10].

VIII. Conclusion

We have presented Vector Runahead, a new microarchitectural method for generating high degrees of memory-level parallelism even for highly complex workloads that feature chains of dependent memory accesses and complex address computation. By dynamically, and speculatively, vectorizing the runahead sequence, we generate a form of runahead execution that brings in many future dependent memory accesses at once, hiding the memory wall by overlapping many vector gather accesses in parallel. We report an average 1.79× performance speedup over a baseline out-of-order architecture with minimal amount of new state added to the processor.

Acknowledgements

We thank the anonymous reviewers and shepherd for their valuable comments. This work is supported in part by the European Research Council (ERC) Advanced Grant agreement no. 741097, the Flanders Research Council (FWO) grant G.0144.17N, and the Engineering and Physical Sciences Research Council (EPSRC) grant reference EP/P020011/1. Additional data related to this publication is available on request from the lead author.

References

[1] S. Ainsworth and T. M. Jones. Graph prefetching using data structure knowledge. In *ICS*, 2016.
[2] S. Ainsworth and T. M. Jones. Software prefetching for indirect memory accesses. In *CGO*, 2017.
[3] S. Ainsworth and T. M. Jones. An event-triggered programmable prefetcher for irregular workloads. In *ASPLOS*, 2018.
[4] S. Ainsworth and T. M. Jones. Software prefetching for indirect memory accesses: A microarchitectural perspective. *ACM TOCS*, 2019.
[5] S. Ainsworth and T. M. Jones. Prefetching in functional languages. In *ISMM*, 2020.
[6] H. Al-Sukhni, I. Bratt, and D. A. Connors. Compiler-directed content-aware prefetching for dynamic data structures. In *PACT*, 2003.
[7] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 1995.
[8] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *ISCA*, 2001.
[9] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, and S. Williams. The landscape of parallel computing research: A view from Berkeley. 2006.
[10] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan. Classifying memory access patterns for prefetching. In *ASPLOS*, 2020.
[11] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel benchmarks – summary and preliminary results. In *Supercomputing*, 1991.
[12] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad. Bingo spatial data prefetcher. In *HPCA*, 2019.
[13] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, 2011.
[14] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *ASPLOS*, 1991.
[15] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM TACO*, 2014.
[16] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout. The load slice core microarchitecture. In *ISCA*, 2015.
[17] M. Cavus, R. Sendag, and J. J. Yi. Informed prefetching for indirect memory accesses. *ACM TACO*, 2020.
[18] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM TODS*, 2007.
[19] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *ASPLOS*, 1992.
[20] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE TC*, 1995.
[21] S. Choi, N. Kohout, S. Pamnani, D. Kim, and D. Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM TOCS*, 2004.
[22] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *ISCA*, 2001.
[23] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *ASPLOS*, 2002.
[24] J. Doweck, W. F. Kao, A. K. y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz. Inside 6th-generation Intel core: New microarchitecture code-named Skylake. *IEEE Micro*, 2017.
[25] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS*, 1997.
[26] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *HPCA*, 2009.
[27] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *ASPLOS*, 2006.

[28] B. Falsafi and T. F. Wenisch. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 2014.

[29] I. Ganusov and M. Burtscher. Efficient emulation of hardware prefetchers via event-driven helper threading. In *PACT*, 2006.

[30] I. Hadade, T. M. Jones, F. Wang, and L. d. Mare. Software prefetching for unstructured mesh applications. *ACM TOPC*, 2020.

[31] T. J. Ham, J. L. Aragón, and M. Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *MICRO*, 2015.

[32] M. Hashemi and Y. N. Patt. Filtered runahead execution with a runahead buffer. In *MICRO*, 2015.

[33] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt. Accelerating dependent cache misses with an enhanced memory controller. In *ISCA*, 2016.

[34] M. Hashemi, O. Mutlu, and Y. N. Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *MICRO*, 2016.

[35] C.-H. Ho, S. J. Kim, and K. Sankaralingam. Efficient execution of memory access phases using dataflow specialization. In *ISCA*, 2015.

[36] A. Jain and C. Lin. Linearizing irregular memory accesses for improved correlated prefetching. In *MICRO*, 2013.

[37] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA*, 1997.

[38] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. In *ASPLOS*, 2002.

[39] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching. In *MICRO*, 2016.

[40] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *MICRO*, 2013.

[41] O. Kocberber, B. Falsafi, and B. Grot. Asynchronous memory access chaining. In *VLDB*, 2015.

[42] N. Kohout, S. Choi, D. Kim, and D. Yeung. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *PACT*, 2001.

[43] R. Kumar, M. Alipour, and D. Black-Schaffer. Freeway: Maximizing MLP for slice-out-of-order execution. In *HPCA*, 2019.

[44] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips. SQRL: Hardware accelerator for collecting software data structures. In *PACT*, 2014.

[45] S. Kumar, N. Vedula, A. Shriraman, and V. Srinivasan. DASX: Hardware accelerator for software data structures. In *ICS*, 2015.

[46] K. Lakshminarasimhan, A. Naithani, J. Feliu, and L. Eeckhout. The forward slice core microarchitecture. In *PACT*, 2020.

[47] E. Lau, J. E. Miller, I. Choi, D. Yeung, S. Amarasinghe, and A. Agarwal. Multicore performance optimization using partner cores. In *HotPar*, 2011.

[48] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann. Livia: Data-centric computing throughout the memory hierarchy. In *ASPLOS*, 2020.

[49] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 2007.

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

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

[52] P. Michaud. Best-offset hardware prefetching. In *HPCA*, 2016.

[53] S. Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys*, 2016.

[54] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slice-processors: An implementation of operation-based prediction. In *ICS*, 2001.

[55] T. C. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, 1994.

[56] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.

[57] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *HPCA*, 2003.

[58] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *ISCA*, 2005.

[59] O. Mutlu, H. Kim, and Y. N. Patt. Address-value delta (AVD) prediction: increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns. In *MICRO*, 2005.

[60] O. Mutlu, H. Kim, J. Stark, and Y. N. Patt. On reusing the results of pre-executed instructions in a runahead execution processor. *IEEE CAL*, 2005.

[61] O. Mutlu, H. Kim, and Y. N. Patt. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro*, 2006.

[62] O. Mutlu, H. Kim, and Y. N. Patt. Address-value delta (AVD) prediction: A hardware technique for efficiently parallelizing dependent cache misses. *IEEE TC*, 2006.

[63] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout. Precise runahead execution. *IEEE CAL*, 2019.

[64] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout. Precise runahead execution. In *HPCA*, 2020.

[65] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA*, 2004.

[66] Q. M. Nguyen and D. Sanchez. Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism. In *MICRO*, 2020.

[67] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki. PrefEdge: SSD prefetcher for large-scale graph traversal. In *SYSTOR*, 2014.

[68] S. Pakalapati and B. Panda. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *ISCA*, 2020.

[69] V. Porpodas and T. M. Jones. Throttling automatic vectorization: When less is more. In *PACT*, 2015.

[70] V. Porpodas, A. Magni, and T. M. Jones. PSLP: Padded slp automatic vectorization. In *CGO*, 2015.

[71] T. Ramirez, A. Pajuelo, O. J. Santana, and M. Valero. Runahead threads to improve SMT performance. In *HPCA*, 2008.

[72] T. Ramirez, A. Pajuelo, O. J. Santana, O. Mutlu, and M. Valero. Efficient runahead threads. In *PACT*, 2010.

[73] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *PACT*, 2004.

[74] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *ASPLOS*, 1998.

[75] A. Seznec. TAGE-SC-L branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[76] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. Efficiently prefetching complex address patterns. In *MICRO*, 2015.

[77] J. E. Smith. Decoupled access/execute computer architectures. In *ISCA*, 1982.

[78] H. Tabani, J. Arnau, J. Tubella, and A. Gonzalez. A novel register renaming technique for out-of-order processors. In *HPCA*, 2018.

[79] H. Takayashiki, M. Sato, K. Komatsu, and H. Kobayashi. A hardware prefetching mechanism for vector gather instructions. In *IA3*, 2019.

[80] K. A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean. Clairvoyance: Look-ahead compile-time scheduling. In *CGO*, 2017.

[81] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *PACT*, 2007.

[82] K. Van Craeynest, S. Eyerman, and L. Eeckhout. MLP-aware runahead threads in a simultaneous multithreading processor. In *HiPEAC*, 2009.

[83] P. H. Wang, J. D. Collins, Dongkeun Kim, B. Greene, Kai-Ming Chan, A. B. Yunus, T. Sych, S. F. Moore, J. P. Shen, and Hong Wang. Helper threads via virtual multithreading. *IEEE Micro*, 2004.

[84] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *ISCA*, 2003.

[85] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin. Temporal prefetching without the off-chip metadata. In *MICRO*, 2019.

[86] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin. Efficient metadata management for irregular data prefetching. In *ISCA*, 2019.

[87] C.-L. Yang and A. Lebeck. A programmable memory hierarchy for prefetching linked data structures. In *ISHPC*, 2002.

[88] X. Yu, C. J. Hughes, N. Satish, and S. Devadas. IMP: Indirect memory prefetcher. In *MICRO*, 2015.

[89] C. Zhang, Y. Zeng, J. Shalf, and X. Guo. RnR: A software-assisted record-and-replay hardware prefetcher. In *MICRO*, 2020.

[90] D. Zhang, X. Ma, M. Thomson, and D. Chiou. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. In *ASPLOS*, 2018.