Vector Runahead for Indirect Memory Accesses

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

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

***Abstract*—Vector Runahead delivers extremely high memory-level parallelism even for chains of dependent memory accesses with complex intermediate address computation, which conventional runahead techniques fundamentally cannot handle and therefore have ignored. It does this by rearchitecting runahead to use speculative data-level parallelism, rather than work-skipping, as its primary form of extracting more memory-level parallelism in runahead mode than a true execution can, which we hope will bring about an entirely new dimension for high-performance processors.**

***Index Terms*—Microarchitecture, runahead execution, prefetching, speculative vectorization, indirect memory accesses**

I. Introduction

Many modern-day workloads are poorly served by current out-of-order superscalar cores, since they feature sparse, indirect memory accesses [3] characterized by high-latency cache misses that are unpredictable by today's stride prefetchers [6]. Despite large reorder-buffer and issue-queue resources, superscalar cores running these applications have run out of steam, spending the majority of their time stalled since they cannot capture the memory-level parallelism necessary to hide today's memory access latencies.

Vector Runahead rearchitects runahead execution to use a new method of generating memory-level parallelism. Rather than work-skipping [8], Vector Runahead extracts *memory-level parallelism* as a speculative form of *data-level parallelism*: it groups together independent loads from many different iterations of the same code, allowing them to all follow different sequences of *dependent* loads *independently*. It further improves throughput by running these newly grouped sequences as vector operations: even when the workload itself is not vectorizable, the prefetching effect from the runahead, which need not be perfectly accurate, is likely to still exhibit data-level parallelism.

On a variety of graph, database and high-performance computing workloads, Vector Runahead improves performance by 1.79× compared to a baseline out-of-order processor with a stride prefetcher. Relative to the state-of-the-art Indirect Memory Prefetcher (IMP) [12] and Precise Runahead Execution (PRE) [9], Vector Runahead improves performance by 1.49× on average. The fundamental reason for this significant performance improvement is illustrated in Figure 1: PRE is unable to accurately prefetch the majority of indirect memory accesses, unlike Vector Runahead.

Vector Runahead, A. Naithani, S. Ainsworth, T. M. Jones and L. Eeckhout, In Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA), June 2021.

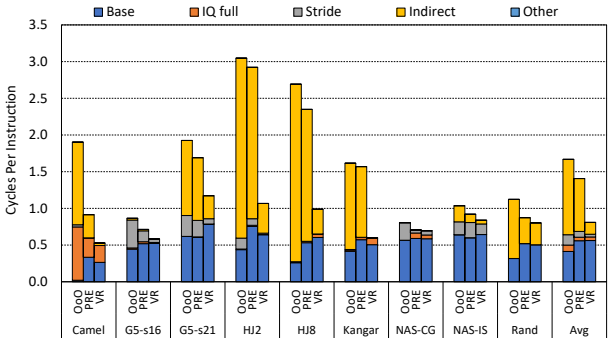


Fig. 1: CPI stacks for the baseline out-of-order (OoO) core, Precise Runahead Execution (PRE) and Vector Runahead (VR). The memory component is broken down and attributed to striding loads and indirect dependent-chain loads. *The previous state-of-the-art runahead cannot prefetch the majority of indirect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

II. Existing Runahead Techniques

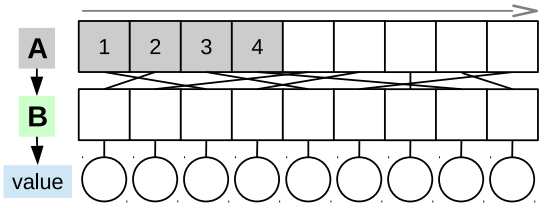
While specialized accelerators are one solution, and programmable forms of prefetching another [1], the ideal solution would be a pure-microarchitectural technique that could achieve the same benefits without the need for recompilation. Hardware prefetchers can pick up a variety of memory-access patterns, but to achieve the instruction-level visibility necessary to calculate the addresses of complex access patterns in today's workloads [1], one must operate within the core, instead of within the cache. Runahead execution [8, 9] is the most promising technique to achieve this.

The promise of runahead execution is that the core can continue to perform useful work even whilst stalled on a long-latency cache miss, by calculating addresses and prefetching data for future memory accesses. By speculatively issuing multiple independent memory accesses, runahead execution significantly increases memory-level parallelism (MLP), ultimately improving overall application performance.

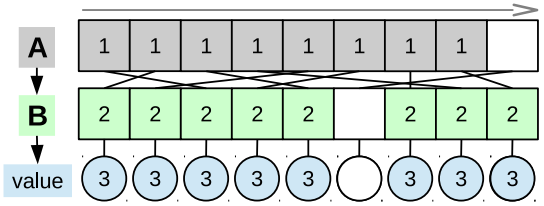
However, conventional runahead comes unstuck by the very mechanism it uses to generate MLP. First, by skipping over loads for which the data source is not yet ready, it is unsuitable for today's complex indirection patterns that consist of chains of dependent load misses. Second, conventional runahead is limited by both the processor's front-end (fetch/decode/rename) width and available back-end resources (issue queue slots and physical registers) [9]. What is needed is a technique that can overcome the limitations of a processor's resources to generate massive amounts of memory-level parallelism and follow chains of dependent loads to completion, prefetching all data required for many memory accesses in the future. Vector

```
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) Precise Runahead Execution (PRE) [9] 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) Vector Runahead vectorizes memory accesses along the memory dependence chain whilst in runahead mode. Multiple accesses to A happen in parallel, followed by parallel accesses to B, followed by parallel data-value reads. 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. This delayed termination condition delivers higher performance by extracting more MLP than an immediate return to normal mode.

Fig. 2: Vector Runahead versus Precise Runahead Execution (PRE) [9] on 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.*

Runahead is that technique.

III. Vector Runahead

The key insight behind Vector Runahead is that many indirect memory accesses occur within loops where each iteration follows approximately the same control-flow path, and that this regularity can be exploited through parallel execution of multiple iterations simultaneously. Speculative vector execution of multiple future loop iterations is possible and safe, even when the original workload is not vectorizable, since the results will be discarded once Vector Runahead is terminated and normal execution resumes.

Vector Runahead addresses the limitations described above in three ways, as illustrated in Figure 2. First, it deliberately waits for the results of currently unavailable loads, rather than invalidating and skipping them, which enables Vector Runahead to prefetch entire load chains but causes the technique to quickly exhaust its backend out-of-order resources and thus stall on waiting for these intermediate results. Second, to fix this, Vector Runahead vectorizes the runahead instruction stream by reinterpreting scalar instructions as vector operations to generate many different cache misses at different offsets. This means that despite executing many future iterations of a loop at once, Vector Runahead only requires the processor resources (both front-end and back-end instruction slots) of a single iteration. In effect, this virtually increases the effective fetch/decode bandwidth during runahead mode by issuing independent operations both in quick succession and merged together into single instructions. Third, it issues multiple rounds of these vectorized instructions through our schemes of vector unrolling and pipelining to speculate even deeper and increase the effective runahead memory bandwidth even further. This has the effect of installing huge numbers of independent loads next to each other in the issue queue and reorder buffer, avoiding the need for out-of-order structures of unbounded size. Altogether, this means that, while vector runahead must wait for dependent loads rather than skipping them, it waits on a huge number of them at once, finally allowing the achievement of extreme memory-level parallelism even on complex workloads.

IV. Microarchitecture Details

We now describe Vector Runahead's required changes to the processor pipeline, as illustrated in Figure 3.

A. Initiating 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 reorder buffer (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). This marks the *entry* to runahead mode. After entering runahead mode, the processor continues to fetch, decode, and execute future instructions. We use a stride detector [6] to find regular access patterns in the code that can be used as 'induction variables' to produce speculative vectorized copies of code. The detector also keeps track of the last dependent load (known as the 'terminator') on the striding load. Entry to *vector-runahead mode* begins when we decode a striding load. We vectorize the striding load, followed by the sequence of instructions depending on it. We call the dependent instructions between two dynamic instances of a striding load an *indirect chain*.

B. Detecting Indirect Chains

We use a *taint vector (TV)* to detect the indirect chains depending on a striding load. The TV 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*).

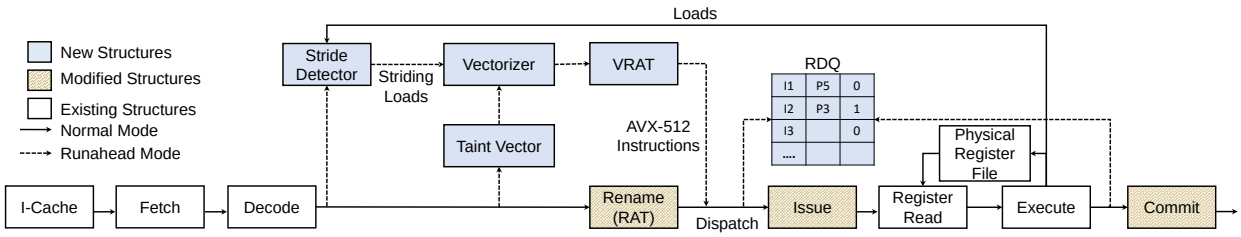


Fig. 3: Processor pipeline for vector runahead execution.

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. 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.

C. Vectorizing Instructions

A microprogrammed routine vectorizes the indirect chain. 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. Regardless of input bit width, eight scalar operands are fit in this 512-bit vector, such that we can operate on any data size up to 64 bits. We assume that each vector instruction uses 512-bit vector registers (similar to Intel's 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.

The renamed instructions are dispatched to the processor back-end where they are executed speculatively. 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 a simpler register-deallocation queue [9] to handle register availability.

D. Vector Unrolling and Pipelining

To cover more iterations of the indirect chain, we can alternatively generate more than one vector instruction for each scalar instruction in the chain. Depending on the amount of back-end resources available, the generated vector instructions can be dispatched to the processor back-end in two ways. First, through *vector unrolling* (Figure 4(b)), we can dispatch vector instructions in multiple rounds. For example, we could dispatch $U \times 8$ copies of a loop by issuing the first eight in a single vectorized copy of the instruction stream in round 1, then repeating the process $U - 1$ times, where U is the *unroll depth*. Second, through *vector pipelining* (Figure 4(c)), we can dispatch all vector instructions for each scalar instruction before dispatching P, the *pipeline width*, vector instructions for the next instruction in the indirect chain. When the amount of back-end resources is limited, vector unrolling is the preferred technique as the processor back-end does not stall due to lack of available resources to process vector instructions. Vector pipelining, on the other hand, delivers better performance when the back-end has sufficient resources to simultaneously process a large number of vector instructions. A processor microarchitecture can be tuned to dynamically select one of the two techniques for higher performance depending on the availability of back-end resources.

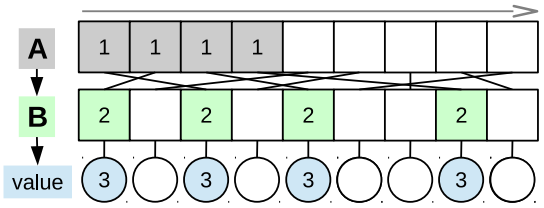
Since we can generate multiple vector instructions for each scalar instruction of the indirect chain, each scalar architectural register first needs to be mapped to multiple vector architectural registers, followed by mapping each vector architectural register to a vector physical register. The complete process of renaming from a scalar architectural register to a vector physical register is accomplished with the help of the *vector register allocation table (VRAT)*, which maintains P, the *vector pipelining width*, 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 uses 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.

E. Control Flow

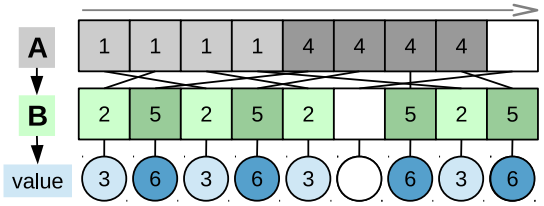
All vector lanes follow the same pattern of control flow, apart from when there is divergence between the lanes in vector-runahead mode when they meet a branch instruction. A micro-op converts scalar branches into a predicate mask for the eight vector lanes. Since Vector Runahead need not cover all code, we 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-flow path.

F. Terminating Runahead

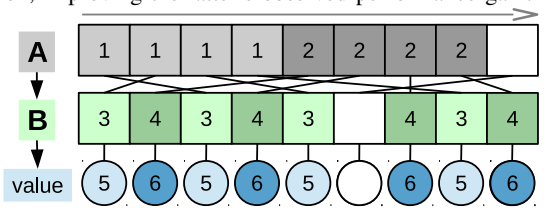
Vector-runahead mode terminates when any of the following four conditions is satisfied: (1) we encounter a dynamic



(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.

instance of the initial striding load again; (2) we encounter, and issue, the *terminator*: the PC identified by the stride detector 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 we dispatch multiple rounds of vectorized instructions in vector unrolling, we re-enter vector-runahead mode immediately, with the next striding load issuing vector gathers again. This is repeated until we have issued all the rounds and only then is normal execution resumed. 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.

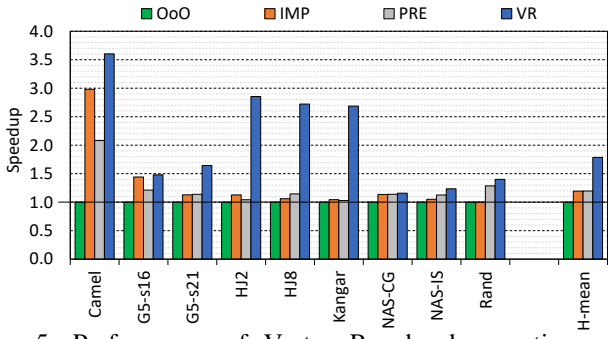


Fig. 5: Performance of Vector Runahead execution on a baseline Intel Skylake-style out-of-order core implemented in Sniper [5]. *Vector Runahead yields a 1.79× and 1.49× harmonic mean speedup compared to the baseline OoO core and PRE (and IMP), respectively.*

G. Hardware Overhead

Vector Runahead requires only modest changes to the processor pipeline, including the stride detector, taint vector, and VRAT. The RDQ is already used by PRE [9]. When put together, the total hardware overhead of Vector Runahead relative to a baseline out-of-order core is limited to 1.3 KB, versus 1.24 KB for PRE.

V. Evaluation

We compare the following microarchitectural mechanisms, all implemented in Sniper [5]:

- Out-of-Order (OoO): Baseline out-of-order core based on Intel's Skylake, with hardware stride prefetcher.
- Precise Runahead Execution (PRE): The state-of-the-art runahead execution technique, as proposed by Naithani et al. [9]. 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. [12]. IMP is attached to the L1 D-cache, and detects indirect access patterns starting from striding memory accesses.
- Vector Runahead (VR): The vector-runahead mechanism proposed in this paper, assuming an *unroll length* U of 8 and *pipeline depth* P of 8.

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), graph and database workloads evaluated in previous work on programmer- and compiler-managed prefetching mechanisms [1, 2].

The 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). We refer to the ISCA 2021 conference paper for details regarding the experimental setup and various sensitivity analyses.

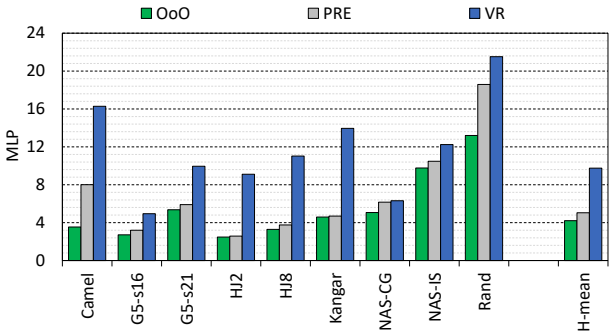


Fig. 6: Memory-level parallelism measured in terms of MSHR entries utilized per cycle if at least one is allocated. *While PRE improves MLP by 1.2×, vectorizing indirect chains generates 2.3× more MLP than an OoO core.*

Figure 5 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, while fetching in all loads within dependent sequences, and without fetching irrelevant data.

Vector Runahead achieves higher performance by three main mechanisms. The most important is the software-pipelining effect that reordering of load instructions provides, in that a large number of misses can be serviced simultaneously. This same reordering when implemented with 64 scalar micro-ops instead of 8 vector micro-ops is 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 increasing the effective processor front-end width, and 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 load instruction at the head of the ROB returns, increasing performance to the full 1.79× shown in the graph.

Figure 6 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 (224-entry) 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, which starve the out-of-order core of reorder-buffer and issue-queue resources [2], 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.

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 data cache misses are serviced quickly with or without Vector Runahead. Finally, even though many workloads end up MSHR-constrained within vector-runahead mode, the average MLP is still typically lower than the number of MSHRs available (24 MSHRs at the L1 data cache in our setup): this is because Vector Runahead does not run continuously, and only kicks in when the out-of-order system runs out of resources.

VI. Potential for Long-Term Impact

Vector Runahead promises a transformational performance improvement for some of today's most important and challenging workloads, all in microarchitecture. At a time when other methods for improving single-thread performance are few and far between, we hope that this work will inspire industry. While the performance improvements are significant, the extra hardware is modest. This reinvention of runahead execution, to be based on speculative data-level (SIMD) parallelism rather than work-skipping as its primary method for hiding memory latency, could be a fundamental building block for many new techniques both inside and outside the core.

Tomorrow's processors will be able to natively support extreme memory-level parallelism, even down complex chains. The recent scaling up of other parts of the microarchitecture, such as highly parallel page-table walkers, means that processors will be able to exploit these benefits to the fullest. In turn, we expect processors to adapt their configurations to accommodate forms of extreme memory-level parallelism as a result: by finally making sparse workloads bandwidth-bound instead of latency-bound, we expect that conventional processors will move to higher-latency, higher-bandwidth memory.

Vector Runahead is a qualitative departure from prior solutions. In particular, in contrast to software auto-vectorization, Vector Runahead does not require the code to be vectorizable to adequately prefetch data into the cache. In contrast to prior runahead techniques, Vector Runahead presents a solution for achieving memory-level parallelism down complex dependent memory chains. In contrast to prior pre-execution and helper-thread techniques, Vector Runahead needs no separate thread, no separate execution units, and neither programmer

nor compiler support. Moreover, Vector Runahead can follow dependent chains, unlike pre-execution and helper threads. In contrast to software prefetching, Vector Runahead is a pure microarchitecture solution, requiring no changes to the binary or source code, while being able to freely vectorize sequences of instructions that would cause software prefetchers to fault. In contrast to hardware prefetching, Vector Runahead operates within-core, allowing it to cover arbitrary memory-indirection depths with complex address calculation, as needed in many workloads [4]. In fact, as we have explored and demonstrated in our ISCA 2021 paper, Vector Runahead provides significant performance improvements for modern-day workloads with complex indirect memory-access patterns from a wide variety of application domains including graph analytics, database, and high-performance computing.

Note further that, while Vector Runahead fundamentally exposes more memory-level parallelism than out-of-order execution, it is not fundamentally reliant upon out-of-order execution. At a time when both out-of-order execution [7] and advanced prefetchers [10] have both been exposed for their inadequacies around security, Vector Runahead proposes a solution for the indirect memory accesses these countermeasures restrict [11] that is reliant on neither out-of-order execution nor out-of-core prefetching. It can preserve secure control flow by being an in-core technique and even despite being speculative itself. We believe that this could finally make such countermeasures [11], and even in-order cores, palatable without severe penalty.

VII. Conclusion

Vector Runahead delivers on what runahead techniques were always designed for, but could never really provide: true latency tolerance for CPUs without out-of-order resources needing to scale to unbounded dimensions, even for emerging workloads with long and complex chains of dependent memory accesses. We believe that Vector Runahead provides an opportunity for transformative improvements in single-thread performance, favoring processor designs optimized for memory-level parallelism rather than being hampered by latency.

BIOs

Ajeya Naithani is a postdoctoral researcher at Ghent University, Belgium. His research interests are in the area of computer architecture with an emphasis on designing novel techniques to improve performance, energy-efficiency, and reliability of modern processors. He received the PhD degree in computer science engineering from Ghent University in 2019. Contact him at ajeya.naithani@ugent.be.

Sam Ainsworth is a Lecturer in Systems and Hardware Security at the University of Edinburgh. His research looks at runtime, systems and hardware security, along with architectural and compiler techniques for data prefetching in software and hardware, and efficient techniques for hardware error detection and correction. He received a PhD (2018) and BA (2014) in Computer Science from the University of Cambridge. Contact him at sam.ainsworth@ed.ac.uk.

Timothy M. Jones is a Reader in Computer Architecture and Compilation at the University of Cambridge. His research interests span compiler and microarchitectural schemes for performance, reliability and security, especially focused on tackling challenges using different forms of parallelism. He received a PhD in Informatics from the University of Edinburgh in 2006. Contact him at timothy.jones@cl.cam.ac.uk.

Lieven Eeckhout is a Full Professor at Ghent University, Belgium. His research interests include computer architecture performance analysis and modeling, and CPU/GPU microarchitecture and resource management. He received a Ph.D. degree in computer science engineering from Ghent University in 2002. He is an IEEE and ACM Fellow. Contact him at lieven.eeckhout@ugent.be.

References

[1] S. Ainsworth and T. M. Jones. An event-triggered programmable prefetcher for irregular workloads. In *ASPLOS*, 2018.

[2] S. Ainsworth and T. M. Jones. Software prefetching for indirect memory accesses: A microarchitectural perspective. *ACM TOCS*, 2019.

[3] 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.

[4] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan. Classifying memory access patterns for prefetching. In *ASPLOS*, 2020.

[5] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM TACO*, 2014.

[6] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *ASPLOS*, 1992.

[7] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[8] 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.

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

[10] J. R. S. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher. Opening pandora's box: A systematic study of new ways microarchitecture can leak private data. In *ISCA*, 2021.

[11] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *MICRO*, 2019.

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