Scalar Vector Runahead: Removing the Shackles of Indirect Memory Chains on In-Order Cores

Jaime Roelandts, *Ghent University, B-9000, Ghent, Belgium*

Ajeya Naithani, *TU Eindhoven, 5612 AZ, Eindhoven, The Netherlands*

Sam Ainsworth, *University of Edinburgh, EH8 9AB, Edinburgh, U.K.*

Timothy M. Jones, *University of Cambridge, CB3 0FD, Cambridge, U.K.*

Lieven Eeckhout, *Ghent University, B-9000, Ghent, Belgium*

Modern processors often face the memory wall as a bottleneck, an exacerbated problem for stall-on-use in-order cores. Despite this limitation, there is growing demand for energy-efficient in-order cores due to privacy and sustainability concerns. Scalar vector runahead (SVR) provides an elegant solution by extracting high memory-level parallelism through piggybacking on existing instructions executed on the processor that lead to future irregular memory accesses. SVR speculatively executes multiple transient, independent, parallel instances of memory accesses and their instruction chains, by initiating memory accesses from many different values of a predicted induction variable. This approach moves mutually independent memory accesses next to each other to hide dependent stalls. With a hardware overhead of only 2 KiB and without the need for hardware vector extensions, SVR delivers 3.2× higher performance than a baseline three-wide in-order core inspired by an Arm Cortex A510, and 1.3× higher performance than an out-of-order core, while halving energy consumption.

Graph analytics, database, and high-performance computing workloads exhibit chains of dependent instructions containing irregular indirect memory accesses. To execute these workloads efficiently, one has to overlap as many of their cache misses as possible to hide memory access latency. The traditional method of overlapping execution relies on out-of-order (OoO) superscalar processors, which use a reorder buffer (ROB) to find independent work. However, even today's largest OoO cores,[2] with increasingly large ROBs, struggle to find enough independent work to achieve good performance on these challenging workloads. Moreover, traditional prefetchers struggle with these workloads due to their irregular memory access patterns.

In-order cores suffer from indirect load stalls even more than their OoO counterparts. A typical stall-on-use in-order superscalar core stalls on the first instruction that depends on a long-latency load, blocking all future memory accesses. Figure 1 illustrates this issue: Although an OoO core suffers from low performance for these benchmarks, the in-order core spends approximately 2.5× more cycles waiting for memory [dynamic random-access memory (DRAM)].

The goal of scalar vector runahead (SVR)[1] is to generate high memory-level parallelism (MLP) on in-order cores for applications with complex chains of memory accesses. SVR achieves this by creating transient

Digital Object Identifier 10.1109/MM.2025.3577524
Date of publication 9 June 2025; date of current version 12 September 2025.

replicas of instructions at issue, such that complex addresses can be speculatively evaluated on the fly for many future chains of dependent loads. These transiently executed replica loads serve as prefetches for future normal execution. The replicated instructions execute in parallel, which increases MLP and thereby performance. SVR includes several optimizations to keep hardware and run-time overhead under control while retaining simple, strictly in-order execution.

PRIOR SOLUTIONS

There have been solutions to improve the performance of applications with indirect memory accesses on OoO cores. However, they are an overkill for power-efficient in-order cores, requiring resources such cores do not have.

Vector runahead (VR)[3] takes chains of instructions, and speculatively vectorizes them in parallel. It starts by vectorizing the first striding load it encounters after the ROB is filled with instructions, followed by vectorizing all of the stride load's dependents. However, VR has three limitations. First, it requires a full ROB in order to activate, which is increasingly rare with the growing size of ROBs seen in modern-day OoO cores.[2] Second, VR stalls normal execution when in runahead prefetching mode. Third, it prefetches a fixed number of future iterations of a loop (for example, 64) at a time. This often results in either overfetching, which pollutes the caches, or underfetching, which leaves performance on the table.

Decoupled vector runahead (DVR)[2] is a solution to those issues. First, it introduces a subordinate hardware thread that runs decoupled and concurrently with the main thread. Second, DVR predicts the number of prefetches and therefore generates only accurate prefetches; it performs a discovery pass over the first iteration of a loop to predict the number of its upcoming iterations. Third, decoupling enables the subordinate thread to manage its own control flow by allowing it to execute divergent paths before reconvergence. Fourth, it can generate prefetches for multiple nested loops.

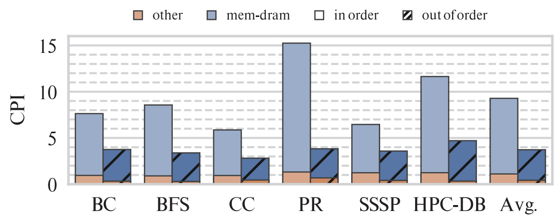

FIGURE 1. An in-order core spends on average 8.17 cycles per instruction (CPI) on DRAM, compared to 3.3 CPI on an OoO core.

Unfortunately, VR and DVR are impractical for power-efficient in-order cores as they rely on expensive wide vector execution units, hundreds of preexisting physical vector registers, and a dedicated execution context (for DVR), which are infeasibly large.

SVR

SVR eliminates the need for vector units and dedicated execution contexts by introducing a *piggybacking* technique: SVR executes the vectorized instructions as sets of replica scalar instructions executed in lockstep with each original instruction, hence the name *scalar vector* runahead. Once a striding load is marked to be vectorized, SVR replicates that instruction *n* times ($n = 16$ in our default SVR-16 configuration), sets the appropriate stride for each replicated instruction, modifies its operands to point to a register in a speculative register file, and issues them speculatively. Instructions that depend on the striding load (directly or indirectly) also get vectorized as scalar-vector instructions. This effectively vectorizes multiple chains of dependent instructions for parallel execution; the dependent loads in each of these chains can hence be prefetched in parallel. Furthermore, speculatively generated instructions are executed concurrently with the main thread to minimize run-time overhead, i.e., normal program execution continues while the speculatively generated scalar vectors prefetch data for future instances of the dependent instruction chain.

SVR INCLUDES SEVERAL OPTIMIZATIONS TO KEEP HARDWARE AND RUN-TIME OVERHEAD UNDER CONTROL WHILE RETAINING SIMPLE, STRICTLY IN-ORDER EXECUTION.

SVR includes three key optimizations to further reduce run-time overhead and carefully mitigate incorrect prefetches.

Optimization #1: Loop-Bound Detection

To avoid out-of-bounds, incorrect prefetches, a loop-bound detection technique is used to determine the degree of vectorization or the number of prefetches.

Unlike DVR that requires an expensive discovery pass, SVR passively observes the in-order program behavior to determine the number of future prefetches. Additionally, the prefetches are generated while the main thread is executing: When the main thread generates a memory access, SVR simultaneously predicts and generates a number of future memory accesses. SVR predicts the number of prefetches using a tournament predictor, which compares a register-based and an exponentially weighted moving average (EWMA) prediction. This new prediction technique handles the commonly observed cases of contiguous loops as well, which occur when a loop starts at the next striding address where the previous loop stopped. The tournament predictor rewards the technique that is the closest to the maximum that could have been accurately prefetched.

Optimization #2: Register Recycling

Unlike VR and DVR, which assume a large preexisting vector register file to keep track of multiple renamings per scalar register, SVR maintains a dedicated small register file, called the speculative register file (SRF), to track destination values of just the most recent vectorized scalars. Each register in the SRF can hold as many scalars as the maximum width of the SVR implementation (16 scalars by default). Incorporating the same number of registers in SRF as the number of architectural registers would be quite expensive. Therefore, to incur low area and energy overheads in SVR, we maintain a small number of registers (eight in our setup) in the SRF and use a least-recently used (LRU) policy to remap registers to new scalars.

Optimization #3: Last Indirect Load

There is no performance benefit from vectorizing instructions that depend on the last indirect load in the chain. The last indirect load typically accesses memory, and waiting for it to return to enable the vectorization of its (nonmemory) dependents only slows down the execution. Therefore, in SVR, once the last indirect load in the chain has been issued, the process of vectorization stops. A bit in the stride detector detects the last indirect load in the chain (see Figure 2).

MICROARCHITECTURE

SVR has three distinct modes of execution. In *normal* mode, the core executes the program and is eligible for initiating SVR. Upon detecting a striding load, the core initiates SVR and enters *piggyback runahead mode*, which vectorizes the dependency chain of the striding load. Once the entire dependency chain has been vectorized, the core enters *waiting* mode, which prevents the core from generating overlapping prefetches and thus wasting expensive scalar compute resources. Waiting mode terminates either when a predetermined number of loop iterations—based on loop bound detection—have passed the issue stage of the pipeline, or the core detects another striding load for which it can generate timelier prefetches. Figure 2 provides an overview of the SVR microarchitecture, and we now briefly discuss the working of its different components and their interactions.

Initiating SVR

The stride detector identifies loads with striding memory access patterns. It scans all of the instructions passing through the issue stage of the in-order pipeline. Upon detecting a load instruction, the stride detector compares its currently accessed address to the previous address; a stride is the difference between the two addresses. A load is marked as striding if the same stride repeats more than a threshold number of times. At issue, a striding load initiates SVR, which generates prefetches for the striding load and the chain of its dependent instructions.

The head striding load register (HSLR) keeps track of the current striding load instruction. Its main purpose is to detect when SVR has completed the vectorization for one iteration of the loop. Therefore, vectorization stops upon encountering the same striding load again; an earlier termination of vectorization is possible if the core encounters the last indirect load.

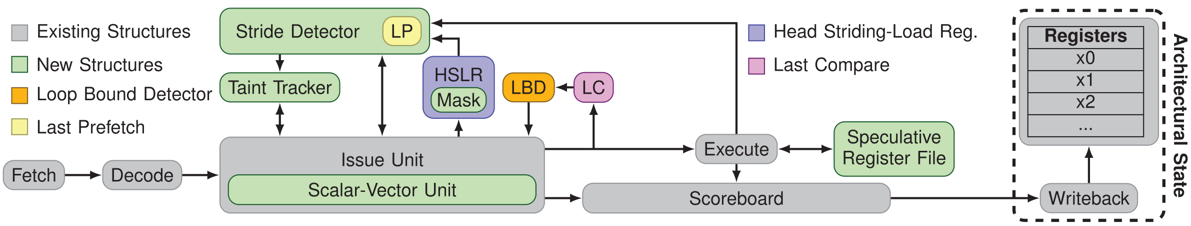

FIGURE 2. SVR's microarchitecture with a stall-on-use in-order core as the baseline.

SVR marks every striding load encountered so far using a *seen* bit in the stride detector, and resets this bit when it finds the load in the HSLR again. However, the core is potentially executing a more inner loop if it finds a striding load for which the seen bit is already set. Retargeting SVR to such an inner loop for vectorization offers opportunity for generating timelier prefetches than the (outer) loop that started with the HSLR. Therefore, SVR switches to the more inner loop for vectorization when this occurs.

Tracking Memory-Access Chains

Once SVR is initiated, the destination register of the striding load is mapped to a vector register in the SRF through the taint tracker (TT) which incurs two main responsibilities. First, it keeps track of the dependency chain of the striding load for vectorization. This can be detected by examining the source registers of a new instruction: If at least one of the registers is marked, then the instruction is part of the chain, and its destination register is marked as well. Second, the TT manages the mapping between architectural registers and vector registers, recycling registers when all vector registers have been mapped to an architectural register. Specifically, the TT uses the LRU policy to remap a vector register to a new architectural register, unmapping the previously mapped architectural register in the process. However, if an unmapped register is later used as a source, the TT does not vectorize that instruction further, as the source values have been discarded.

Loop Bound Prediction

The last compare register (LC) and loop bound detector (LBD) structures determine *n* or the loop bound. They track the last compare instruction before a striding load, which usually holds information about the upcoming number of iterations of the loop. The LC stores details about the latest compare instruction, including its instruction pointer, register identifiers, and register values. The LBD mirrors this information but updates only when a branch instruction jumps back to an earlier instruction pointer than the striding load.

When predicting *n*, the LC holds values from the previous iteration of the loop while the current values are in the LBD. By comparing LC and LBD, SVR can estimate the number of future iterations of the loop. Each time a striding load is issued, the tournament predictor refines its estimate. Simultaneously, the EWMA updates by taking seven-eighths of its previous value and adding one-eighth of the expected value.

Speculatively Vectorizing the Instruction Stream

The job of the scalar vector unit (SVU) is to replicate and execute instructions marked by the TT. During replication, the source registers are checked to see whether they are mapped to a vector register. If so, the corresponding architectural register will be substituted with the mapped vector register at the corresponding index for the replicated instruction. Similarly, the destination registers are substituted. For each marked instruction, the SVU generates *n* replicas, *n* being the size of the vector. These new instructions are sent to the execution units, and a counter in the scoreboard keeps track of their completion. A marked instruction is retired from the in-order pipeline only when all its replicas have finished their execution.

Terminating SVR

The stride detector keeps track of the expected last memory address to be issued by the striding load. In waiting mode, the core checks that the memory addresses accessed by the normal execution of the striding load do not surpass the expected last address. The core terminates waiting mode, and SVR in turn, in case the real execution digresses from the predicted execution.

KEY RESULTS

Figure 3 reports harmonic-mean speedups (left) and energy (right), both normalized to a three-wide stall-on-use in-order baseline. Relative to the OoO core, the low performance on the in-order core is due to its stall-on-use nature. The dependent of a memory access in one iteration of a loop blocks the (possibly) independent striding memory accesses in future iterations. The OoO core, on the other hand, can issue independent striding accesses until its ROB fills up. This allows

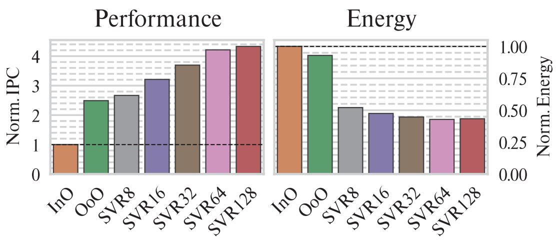

FIGURE 3. Average speedup (harmonic mean of IPC) and whole-system (CPU plus memory) energy for various lengths of SVR (16 default) versus a three-wide in-order core baseline, and a three-wide OoO core. IPC: instructions per cycle.

generating a small amount of MLP on the OoO core by launching a limited number dependent-memory-access chains once the strides return. SVR can issue an even higher number of memory accesses than can fit in the ROB of the OoO core. Overall, SVR performs better than both the in-order and the OoO cores, by 3.2× and 1.3×, respectively. The performance of SVR increases with increasing vector size, and SVR-128 extends speedup versus an OoO core to 1.7× (4.2× relative to in-order), demonstrating that SVR is scalable.

While providing a substantial performance speedup, SVR-16 also decreases energy consumption by 53%. Moreover, energy consumption stays this low even when larger vector sizes are used. SVR achieves this while incurring a hardware overhead of only 2.17 KiB (SVR-16) and 9 KiB (SVR-128).

Figures 4 and 5 present the effectiveness of the loop-bound prediction technique. The key observation is that loop-bound prediction improves accuracy from 88% to almost 100%, emphasizing that SVR performance is not degraded by incorrect prefetches. Furthermore, SVR covers 83% of all loads on average, demonstrating its wide effectiveness.

Sensitivity analysis also shows that SVR does not require a large number of resources to perform well.[1]

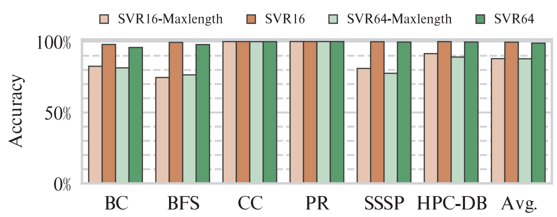

FIGURE 4. Accuracy: Proportion of prefetched cache lines accessed by the core within any cache before eviction from the last-level cache. SVR-Maxlength shows SVR without loop-bound prediction.

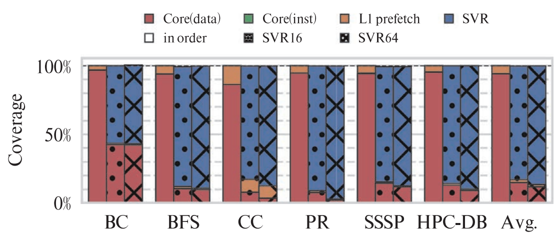

FIGURE 5. Coverage: Proportion of loads that reach the DRAM controller from different origins, normalized to the in-order core; what exceeds 100% is caused by inaccurate prefetches.

Merely eight to 16 miss-status holding registers, two page table walkers, and 25 GB/s of memory bandwidth is sufficient to achieve near-optimal performance that SVR can offer for a vector size of 16.

Issuing loads one by one, instead of relying on existing vector units, does not impact the performance of SVR significantly. Increasing the execution capacity, to allow eight loads per execution unit, only yields an uplift of 3.9%. This is because SVR already saturates memory bandwidth with prefetch requests, so the compute throughput has little impact on performance.

Since SVR does not have a discovery pass as in DVR, the first iteration of the loop-bound prediction might fail, as the LBD and LC do not have up-to-date information. When such an edge case is detected, we evaluate multiple policies: waiting for one iteration, using the maximum length of a vector, or reading the current values of the architectural registers instead of those stored in the LBD. The latter policy yields the best average performance but falls short for benchmarks where a more aggressive technique performs better, such as PageRank (PR) and Connected Components (CC). The EWMA technique can take care of those cases by performing more prefetching across different inner-loop invocations.

LOOKING FORWARD

SVR Targets Emerging Workloads

SVR targets an important and increasingly prevalent class of workloads that consist of multiple chains of dependent loads. Such execution patterns are prevalent in graph analytics and database workloads, which are ubiquitous in social media applications, recommendation systems, key-value stores, and so on. Chains of dependent loads are notoriously hard to prefetch for two key reasons: 1) loads that depend on each other serialize execution and hence cannot be issued in parallel to hide their memory access latency, and 2) chains of dependent loads typically lead to irregular memory access patterns, which are hard to prefetch with traditional prefetching proposals. This is why hardware prefetchers are advised to be turned off for these types of workloads as they are ineffective and even detrimental to performance by creating cache pollution and memory bandwidth congestion.[4]

Enabling Vector Runahead on Low-Power In-Order Cores

SVR greatly lowers the complexity threshold for taking advantage of the new forms of reorder-based runahead introduced by VR[3] and DVR,[2] which fundamentally rely on expensive hardware structures only present in high-end OoO cores to speculatively vectorize chains of dependent loads for prefetching. SVR provides a pathway and fundamental insights toward introducing vector runahead concepts in low-power in-order cores. Piggybacking and intersecting speculative vector instructions with the main execution thread, issued as scalar instructions, enables the vector runahead concept to deliver a substantial performance boost without requiring expensive hardware structures such as vector execution units and large vector register files, as is the case for VR and DVR. Furthermore, adding loop-bound detection, register recycling and last-indirect load determination minimizes run-time overhead and maximizes the performance benefit SVR can achieve. This leads to an overall substantial performance boost over an in-order baseline. Moreover, SVR on top of an in-order core even significantly outperforms an OoO core. Finally, SVR halves energy consumption compared to both in-order *and* OoO cores, making it an energy-efficient optimization.

SVR as a Sustainable Design Option

With a growing demand to run graph analytics and database workloads, we believe there is overwhelming temptation for yet another accelerator to be added. In fact, various graph analytics accelerators have been proposed.[5,6,7] However, SVR provides the opportunity for small cores to do the heavy lifting themselves. The small overhead of SVR allows it to be more area-efficient than a hypothetical accelerator, while still being flexible to run other applications. This is especially important because chip area has a direct correlation to embodied emissions which account for the biggest part of the total environmental footprint of mobile low-power devices.[8] Despite the small hardware overhead, SVR still provides a high performance boost for graph analytics and database workloads running on small in-order cores. The performance gain allows the in-order core to reduce the execution time, much more than the increase in power consumption, resulting in a net reduction in energy consumption, which means much longer battery lifetimes and/or more analysis for the same budget.

SVR Enables Data Privacy-Preserving Edge Computing

Today, the only feasible way for low-power devices to run graph and database workloads is to offload to the cloud. This comes with quality-of-service challenges due to communication delays and interruptions. In addition, there is a growing concern regarding data privacy when interacting with the cloud. Edge computing provides a solution by maintaining and processing data locally. One of the key impediments to running graph and database processing on edge processors though is its limited performance, which SVR overcomes while preserving data privacy.

Potential for Commercial Adoption

Because of the substantial performance, energy and sustainability benefits while incurring minor hardware changes, we expect SVR integration into commercial edge processors to be an easy decision. While this work demonstrated SVR on top of an in-order core, we believe it can as easily be deployed on top of OoO cores, to deliver simple yet effective prefetching opportunities for the most challenging memory-bound workloads — doing so will finally bring runahead techniques to be part of the large family of increasingly specialized prefetching mechanisms in modern cores.[9] We hope this work further sparks research into a wide range of similar, low-complexity techniques designed to target all kinds of memory-bound workloads through finding diverse forms of latent MLP that can be easily discovered through new, subtle ways of interpreting the instructions inside programs.

CONCLUSION

SVR produces speculative prefetches, by piggybacking scalar vector instructions on top of the main execution thread. These scalar-vector instructions are replicas of the issued instructions, and are executed sequentially as scalars to eliminate the need for complex vector execution units. SVR introduces three key optimizations to enhance its efficiency on an in-order core. First, it predicts loop bounds using a tournament predictor instead of relying on a discovery pass, simplifying loop-bound prediction. Second, it minimizes the amount of state required to execute scalar vectors by aggressively recycling vector (physical) registers. Third, it terminates the process of generating scalar-vectors at the last indirect load in a chain, therefore improving performance by waiting for the dependents of the last indirect loads to execute. SVR yields significant speedups on low-power in-order cores and reduces energy consumption by more than half. Overall, we believe that SVR offers a foundational and compelling solution for the growing demand of graph and database workloads at the edge.

ACKNOWLEDGMENTS

This work was supported in part by Research Foundation Flanders (FWO) under Grant G018722N, European

Research Council (ERC) Advanced Grant 741097, and the Engineering and Physical Sciences Research Council (EPSRC) Grant EP/W00576X/1. Additional data related to this publication is available on request from the lead author.

REFERENCES

1. J. Roelandts, A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Scalar vector runahead," in *Proc. 57th IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Piscataway, NJ, USA: IEEE Press, Nov. 2024, pp. 1367–1381, doi: 10.1109/MICRO61859.2024.00101.
2. A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Decoupled vector runahead," in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, New York, NY, USA: ACM, Nov. 2023, pp. 17–31, doi: 10.1145/3613424.3614255.
3. A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector runahead," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, Piscataway, NJ, USA: IEEE Press, Jun. 2021, pp. 195–208, doi: 10.1109/ISCA52012.2021.00024.
4. A. Jain et al., "Limoncello: Prefetchers for scale," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst., Vol. 3 (ASPLOS)*, New York, NY, USA: ACM, 2024, pp. 577–590, doi: 10.1145/3620666.3651373.
5. C.-Y. Gui et al., "A survey on graph processing accelerators: Challenges and opportunities," *J. Comput. Sci. Technol.*, vol. 34, no. 2, pp. 339–371, Mar. 2019, doi: 10.1007/s11390-019-1914-z.
6. L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, New York, NY, USA: ACM, 2014, pp. 255–268, doi: 10.1145/2541940.2541961.
7. O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, New York, NY, USA: ACM, 2013, pp. 468–479, doi: 10.1145/2540708.2540748.
8. L. Eeckhout, "FOCAL: A first-order carbon model to assess processor sustainability," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst., Vol. 2 (ASPLOS)*, New York, NY, USA: ACM, 2024, pp. 401–415, doi: 10.1145/3620665.3640415.
9. A. Pellegrini, "Arm Neoverse N2: Arm's 2nd generation high performance infrastructure CPUs and system IPs," in *Proc. IEEE Hot Chips 33 Symp. (HCS)*, 2021, pp. 1–27, doi: 10.1109/HCS52781.2021.9567483.

JAIME ROELANDTS is a Ph.D. student at Ghent University, B-9000, Ghent, Belgium. His research interests include computer architecture with an emphasis on simulation and graph processing. Roelandts received his M.Sc. degree in computer science engineering from Ghent University. Contact him at jaime.roelandts@ugent.be.

AJEYA NAITHANI is an assistant professor at TU Eindhoven, 5612 AZ, Eindhoven, The Netherlands. His research interests include computer architecture with an emphasis on designing novel techniques to improve performance, energy-efficiency, and reliability of modern processors. Naithani received his Ph.D. degree in computer science engineering from Ghent University. Contact him at a.naithani@tue.nl.

SAM AINSWORTH is a research consultant working in industry and an Honorary Fellow at the University of Edinburgh, EH8 9AB, Edinburgh, U.K. His research interests include architectural and compiler techniques for data prefetching in software and hardware, along with runtime, systems and hardware security, and efficient techniques for hardware error detection and correction. Ainsworth received his Ph.D. degree in computer science from the University of Cambridge. Contact him at sam.ainsworth@ed.ac.uk.

TIMOTHY M. JONES is a full professor in computer architecture and compilation at the University of Cambridge, CB3 0FD, Cambridge, U.K. His research interests include compiler and microarchitectural schemes for performance, reliability, and security, especially focused on tackling challenges using different forms of parallelism. Jones received his Ph.D. degree in informatics from the University of Edinburgh. Contact him at timothy.jones@cl.cam.ac.uk.

LIEVEN EECKHOUT is a full professor at Ghent University, B-9000, Ghent, Belgium. His research interests include computer architecture performance analysis and modeling, CPU/GPU microarchitecture and resource management, and sustainability. Eeckhout received his Ph.D. degree in computer science engineering from Ghent University. He is an IEEE and ACM Fellow. Contact him at lieven.eeckhout@ugent.be.