Evaluating Precise Runahead Execution with Data Prefetching

Matthew Barondeau, Karan Gurazada, Aditya Tewari

***Abstract*—Memory latency is a major performance bottleneck for modern processors, and a variety of methods have emerged to mitigate these long latencies. Two such approaches, data prefetchers and runahead execution, exploit parallelism to request data before the executing thread requests the data. While these two approaches offer impressive performance on their own, there has been little work on comparing prefetchers and runahead execution nor has there been work on analyzing their combined performance. In this paper we show that modern prefetchers outperform precise runahead due to better timeliness. However, combining runahead and these prefetchers does lead to an overall speedup, with the best configuration providing a 13% IPC uplift on the SPEC2006 benchmarks.**

I. Introduction

The memory wall creates a significant bottleneck for modern day applications and hardware. Because of the large latency to access DRAM, hardware designers have worked to reduce these latencies by utilizing some form of prefetching. The basic principle behind prefetching is to predict that a memory request will occur in the future and fetch that data into the cache before the processor needs it. Most prefetching methods have focused on using data from the past access to predict whether to fetch some data, but this approach has limitations that are discussed in section II-A. A novel approach, proposed by Onur Mutlu, is to simply allow the processor to make the prefetches when it would otherwise have been stalled. This approach, called runahead execution, enables the processor to make predictions with more information about program state, but it can have difficulty issuing timely requests.

While both data prefetching and runahead execution have their advantages, no existing paper performs a coherent evaluation of interactions between these two types of prefetching. As a result, it is not possible to know the quantifiable benefit runahead is able to provide over prefetching, or how well the two systems interact. Our goal is to provide a thorough evaluation of runahead execution against existing prefetching techniques in order to better understand the benefits and determine if the two mechanisms provide a mutual benefit running on the same system. Our contributions are:

- To compare runahead execution against other methods of data prefetching.
- To evaluate a system that uses a combination of runahead execution and other methods of data prefetching.
- To understand the performance characteristics of runahead execution.
- To provide a repeatable framework for others to do this performance analysis using scarab.

II. Prior Work

There exist several solutions to mitigate the performance penalty associated with the memory wall, and all the solutions rely on some form of parallelism or locality. There exist several locations for parallelism, either in the form of additional instruction level parallelism like runahead execution, or memory parallelism using prefetchers.

A. Data Prefetching

In an out of order processor, the out of order window can hide small memory latencies, but large latencies can stall the pipeline. To mitigate these stalls, architects commonly use caches to store frequently used data, but often these caches are not sufficiently large or have not seen the data that the program is accessing next. We ran experiments on eight different types of prefetchers designed to handle both regular and irregular access patterns.

1) Regular Access Prefetchers: A regular access pattern is one where the address to be prefetched is at an easily predictable constant or semi-constant offset from the demand access. The simplest regular access prefetcher is the nextline prefetcher [6], which on a demand hit prefetches the cache line with the address immediately following the hit. The stride prefetcher [2] assumes that the line to be prefetched will come at some small offset before or after the demand access and uses past history to find what the value for the stride should be. The PC-localized stride does the same but maintains a different stride for each PC, thus allowing it to track different access patterns by each load instruction. The stream prefetcher used in the IBM POWER4 processor [12] recognizes large groups of incoming lines that are all correlated. Finally, the instruction pointer classifying prefetcher (IPCP) [10] classifies each load instruction as either a constant stride, a complex stride (an access pattern with multiple offsets that usually occur in the same order), or a stream (a long block of consecutive addresses that are usually accessed together). The IPCP is able to target each of these three simple access patterns differently and falls back on the nextline prefetcher when no history or clear classification is available.

2) Irregular Access Prefetchers: Not all access patterns in real programs are regular. Many prefetchers heavily exploit the past access history in order to predict these irregular access patterns. The Global History Buffer (GHB) [9] prefetcher maintains a large record of all demand accesses. By maintaining the overall access history, the GHB prefetcher is able to search for previous occurrences of a load address and prefetch the addresses that had come before it in the past.

The Markov prefetcher [5] models a heavily optimized Markov chain where each address has a most likely successor, which in turn each have a most likely successor. The prefetcher uses the Markov chain probabilities to store the access pattern without the massive overhead of storing history. The Irregular Stream Buffer (ISB) [3] prefetcher keeps track of demand accesses in a structural address space that reflects how correlated lines are. Two lines with disparate physical addresses that are often accessed one after the other will be consecutive in the structural address space, meaning that a nextline prefetcher in the structural address space, which is ISB's operation mechanism, is very effective.

B. Runahead Execution

Rather than relying on memory parallelism or caches, runahead execution extends Out of Order execution to execute speculatively when the processor would otherwise stall. The original variant proposed by Mutlu et al. [7] checkpoints the re-order buffer before allowing the blocking instruction to be cleared and the program to begin executing speculatively. The program is able to continue executing and issuing additional memory requests until the original blocking instruction's dependencies are resolved. At this point, the reorder buffer is reset to the check pointed version and the program begins re-executing the prior core, but since it issued those original memory requests that data should arrive sooner or already be in the caches.

While the baseline runahead execution provides impressive performance gains, it is extremely energy inefficient as you end up executing instructions multiple times with the overall result being faster by doing more work through the run. A new variant, called precise runahead [8] resolves some of this energy inefficiency by storing some of the speculative work in the functional units rather than completely flushing the pipeline back to the checkpoint state. By saving the result, this variant both saves energy and improves performance by not having to re-execute all the speculative instructions.

Runahead execution can provide improvements without high energy or storage requirements needed by other forms of parallelism, but most research up to this point has been on improving the efficiency of the mechanism and making it more practical. Both the original runahead paper and precise runahead look at runahead in isolation or with an extremely small and simple prefetcher. In this project, we look to examine runahead execution's performance in tandem with other memory latency mitigation measures.

III. Expected Benefit

While there exist many works integrating prefetching and caches, some even formally showing an optimal configuration for their combination [4], little work has been done investigating runahead in concert with data prefetching. To the authors' knowledge, papers examining the two resort to extremely small or simple configurations [7] [8] [1], while showing respectable performance gains for these configurations. While the two mechanisms might seem at odds since they are both fundamentally methods of warming up caches for the active process, the difference in trigger mechanism and scope make them compatible and mutually beneficial.

Data prefetchers try to predict upcoming accesses to the cache using prior access information using methods such as address or delta correlation, but these mechanisms rely on the past information being an accurate representation of the future. In contrast, runahead execution makes no assumption about past behavior, but instead uses upcoming program behavior to access data speculatively before executing it normally. Since runahead follows the program exactly and does not rely on any correlation, the future need not look like the past and your runahead acts as a perfect prefetcher. The downside to this approach however, is that your window has to stall before you can begin running in this "perfect prefetcher" mode and you have the program delay before you can execute your program. Furthermore

While runahead can execute whenever the window is stalled, the benefit for runahead is best for longer windows stalls. Since runahead is executing a slice of the program, it needs a sufficiently long time period for the requests to be timely. If the window unblocks quickly, the runahead will not have issued the new instructions very far in advance of the regular access and the benefit is minimal. As a result, the benefit for runahead execution is highest for long stall windows and reduced during periods of short stalls. The pathological case for runahead is a program with many short stalls as runahead will end up speculatively executing a few instructions in advance before they are executed normally. Therefore, if the smaller window stalls are reduced by data prefetching, runahead would be called primarily invoked on long stalls that could not be prefetched, and should allow better performance.

In addition to the benefits that prefetching grants to runahead execution, running ahead should improve prefetcher timeliness. Since runahead is executing during a period when the processor would otherwise be stalled, it is generating new memory addresses that the prefetchers can train on. As a result, the prefetchers can begin issuing new prefetch requests earlier as their trigger event has been moved up.

IV. Methodology

A. Simulator Setup

To evaluate runahead execution and data prefetching, we utilized scarab [11] a cycle-accurate simulator that uses an Intel PIN frontend and a timing backend to model a processor. We chose this simulator due to the availability of a runahead implementation and prior experience with the simulator. In our simulations, we modeled a Kaby Lake processor with the configurations shown in Table I. We selected existing, functional, prefetcher configurations and ran them alongside runahead mode. Scarab relies on physical addresses traces and no threading, so we ran only single core configurations for our workloads. Furthermore, scarab only models two levels of cache hierarchy, so our experiments were restricted to this configuration. For our workloads, we selected SPEC int_speed workloads, but the SPEC2017 benchmarks encountered errors running in scarab, so we report just SPEC2006.

Component	Configuration
L1	32KB, 8 way, 1 cycle delay
L2	1 MB, 16 way, 18 cycle delay
Memory	DDR4_2400R
OOO Window	6 wide issue, 176 ROB entries
Runahead	max runahead length 10000, 2 cycle start delay

TABLE I
PROCESSOR CONFIGURATION

B. Dockerization of Builds

A significant issue with prior work in the runahead space is the lack of comparison to data pre-fetchers. The reason for this disparaty is the clunkiness of the simulator and the amount of time it takes to run a particular simulation. In order to speed up this process we used a dockerized container environment to generate checkpoints, do evaluations, and debug. Repo: https://github.com/AdityaAtulTewari/docker_scarab_runahead_wrapper.git

1) Checkpoint Generation: A container labeled `base` is used to generate checkpoints from both SPEC2006 and SPEC2017. It can be extended with other benchmark suites with modifications to the base Dockerfile and the Makefile section that discusses generating checkpoints.

2) Simulation Evaluations: These evaluations are run by the container labeled `latest`. A container designed for executing different configurations of scarab. This enables you to add new parameters to scarab and modify the simulator. Then you can rebuild the container and use the Makefile to execute the desired benchmarks with the desired parameters. The container is run with the particular benchmarks checkpoints mounted and will spawn a thread per checkpoint. This thread spawning is done to maximize parallelism on each benchmark and on all runs that are desired to be run at the same time.

3) Debug Environment: THe debug environment is a container that can be used to attach to different parts of the scarab simulator and is setup very much like the `latest` environment; it is found in the `debug` container.

This environment is what enables us to seemlessly run scarab without much effort on a machine of any size. Although the system takes some time to setup, and the checkpoints take time to generate, we have made scarab something anyone can setup. Hopefully with this system more people will be able to use it.

V. EVALUATION

A. Evaluating Standalone Systems

We first performed a sweep of prefetcher parameters with the prefetcher configurations we had available. The final configurations we selected were the best performing and are shown in Table II.We ran each prefetcher configuration in isolation and compared their IPC to that of a processor with precise runahead and a baseline no-prefetcher and non-runahead system. The results from the initial sweep are shown in Figure 1. From this sweep, we found that runahead performs worse than our stronger prefetcher conditions, which was the opposite of our expectation. Further compounding the problem was a bug that accidentally left the nextline and stream prefetchers on during our initial runahead results. When these errors were corrected, we found that a sufficiently powerful prefetcher was able to beat a runahead result.

Prefetcher	Configurations
GHB	64K entries, default degree 16, maximum degree 32
Markov	64K Entries, tracks 8 next states
Stride	16K Entries, default degree 8
Stridepc	16K entries, default degree 8
Stream	stream length 64, degree 4 prefetcher

TABLE II
PREFETCHER CONFIGURATIONS

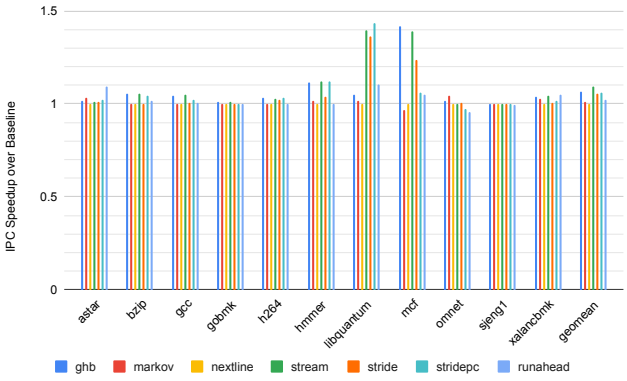

Fig. 1. IPC Speedup over Baseline using Standalone Prefetchers

From our accidental bug, we found that having multiple prefetchers enabled with runahead would lead to significant speedups. To further examine this, we created 5 variants from our best performing prefetchers by combining the two strong variants to run at the same time. When we ran this against runahead, as shown in Figure 2, the performance gains over runahead were even more stark. When we investigated why these performance gains were occurring, we found that prefetchers were greatly increasing the L2 hit rate, while runahead mainly increased L1 accesses. This is because runahead warms up the caches immediately before execution and brings data either into the OOO window or into the L1 cache, while all of our prefetcher configurations target the L2 cache. This is captured in the first two columns of Figure 3 and in the prefetcher only columns.

Since scarab has a relatively high L2 delay and runahead is able to fetch data into the L1 cache, we ran our workloads with an ideal L2 to L1 prefetcher to determine the maximum performance of the prefetcher configurations without runahead execution. When we ran these configurations however, any prefetcher that was in the L1 cache caused the simulation environment to crash, especially with the ideal prefetcher. Since our goal is to determine if runahead execution has a benefit in a system with data prefetchers, and the last level cache is a reasonable place to prefetch, we leave exploring L1 prefetcher interactions with runahead to future work.

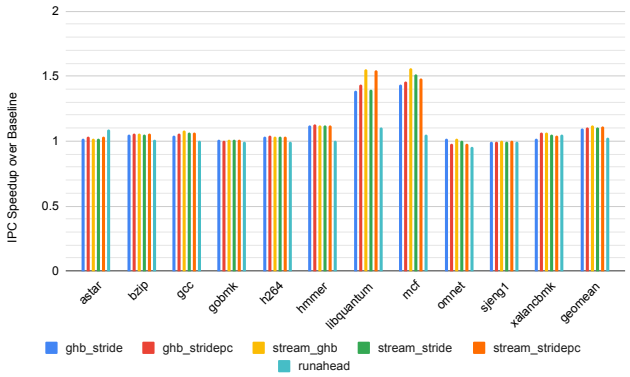

Fig. 2. IPC Speedup over Baseline using Composite Prefetchers

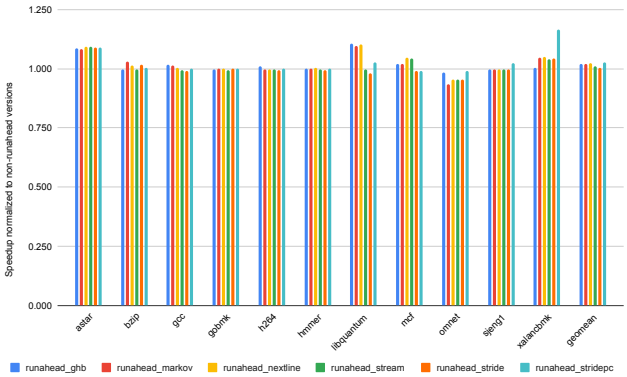

Fig. 4. IPC Speedup of simple prefetchers over non-runahead version

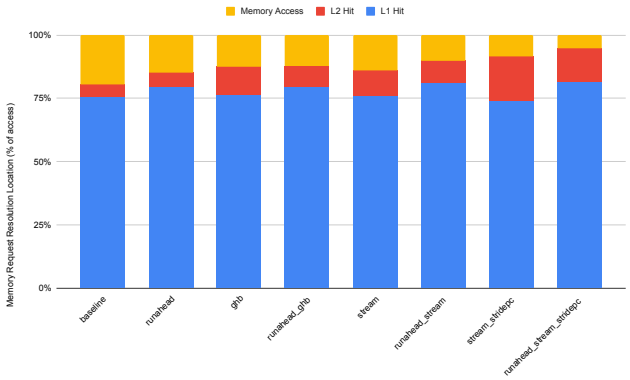

Fig. 3. Memory Access Behavior for SPEC2006 Workloads

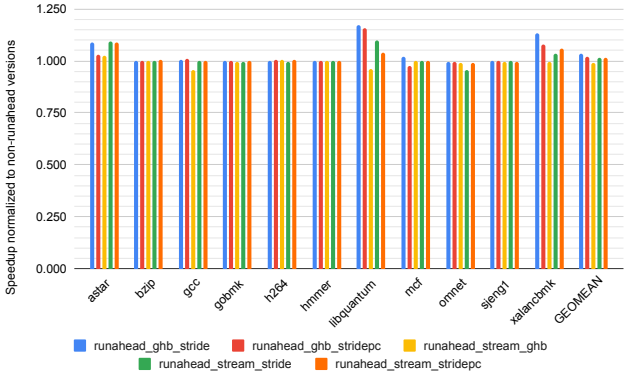

Fig. 5. IPC Speedup of composite prefetchers over non-runahead version

B. Combining Runahead Execution and Prefetchers

With the standalone systems examined, we turned our efforts towards determining if our expected benefit for combining the prefetchers and runahead actually panned out. We first combined the single prefetcher configurations with runahead and ran them on SPEC2006. Their IPC results are normalized to their non-runahead execution times and shown in Figure 4 and the composite prefetchers combined with runahead are shown in Figure 5.

When we combined our prefetchers with runahead execution, we saw a performance increase on average, but a much smaller increase than we were expecting. In general, when we combined runahead with the single prefetchers, we got a speedup of 1.8%, but when we combined with the composite prefetchers, we found that our speedup dropped to 1.6%. While this is a small margin, it was consistent across our benchmarks. With such a small benefit for the cost of implementing runahead, we began investigating why the benefit was small when the two mechanisms seem orthogonal.

While we saw only minor improvements in IPC, we found that combining the prefetchers and runahead dramatically reduced the number of times that the out of order window was stalled. Opposite to our intuition, runahead execution was able to reduce the total number of stalls Figure 6, but the composite prefetchers were able to vastly reduce the number of longer stalls that the runahead mechanism was not able to hide, as shown in Figure 7. When we examined how far each runahead interval was, we found a large variance between workloads, shown in Figure 9. In general, the workloads that performed well with runahead had longer instruction windows, as shown in Figure 1. This is because with a longer interval, runahead execution is able to issue more requests and the timeliness of the requests later in the run can approach that of a prefetcher. When the runahead interval is short however, runahead is only able to execute a few instructions and the timeliness is poor. Furthermore, since runahead is executing instruction traces, it might execute the wrong path and issue an incorrect memory access that will compete with the prefetchers for memory bandwidth.

Instead, runahead execution allows for minor, difficult to prefetch, loads to not fully stall forward progress and the small stalls to be reduced. To verify this, we examined the number of long stalls and total stalls reduced for the composite prefetchers with runahead execution. We define long stalls as stalls of the out of order window that last for greater than 20 cycles. We find that with our standalone configurations, runahead execution does a better job of reducing the total

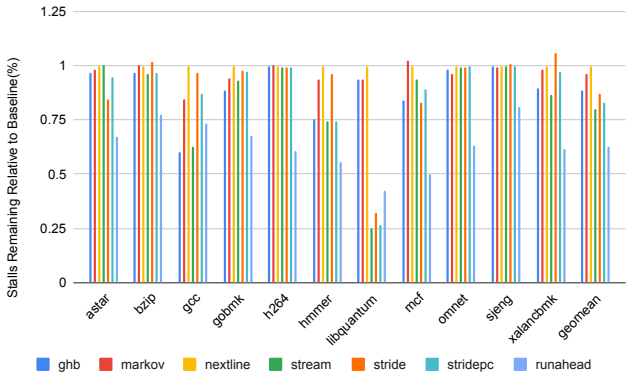

Fig. 6. Stall Reduction relative to Baseline

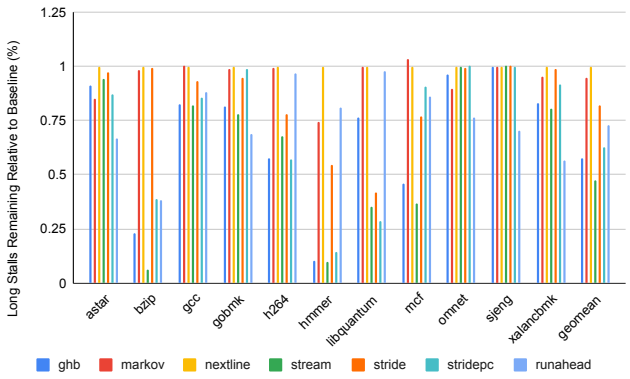

Fig. 7. Long Stall Reduction relative to Baseline

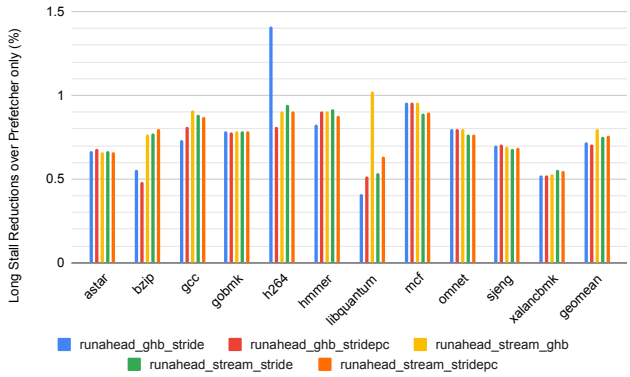

Fig. 8. Additional Long Stall Reduction from Runahead Execution

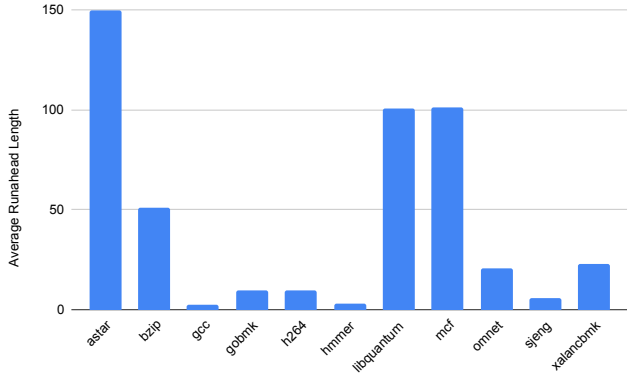

Fig. 9. Average Runahead distance on SPEC2006

number of stalls, but data prefetchers do a better job of covering misses that result in long window stalls as shown in Figure 6 and Figure 7. Since these long stalls are periods where the processor cannot make progress, they are typically long latency operations like a memory access and reducing these stalls will have a greater positive impact on performance than the slow stalls. However, we find that there is still a benefit to integrating prefetching and runahead execution by removing these smalls stalls. We combine the composite prefetchers and the precise runahead mechanism and show in Figure 8 that relative to the non-runahead versions, our combinations can drastically reduce the number of long stalls.

The final speedups for all configurations relative to the baseline are shown in Table III. Our best performing configuration was a composite prefetcher combining a stream prefetcher and a PC-localized stride prefetcher along with runahead execution with a total IPC speedup of 13% over the baseline. The best single performing prefetcher was the Stream prefetcher with a speedup of 9.5% over the baseline. We did not compare runahead with the nextline predictor due to the low change in performance from the nextline predictor.

Overall we found that runahead execution can achieve a speedup when combined with data prefetching, but the speedup is minor and most of the benefit to the workloads come from data prefetchers. Runahead execution performs best when there are long runahead periods, but all of our workloads showed very small average runahead intervals, which inhibit the ability of runahead to be timely when issuing new memory requests. With longer stalls, such as with TLB misses runahead might get additional speedup, but scarab lacks the ability to model virtual memory so we cannot explore this avenue. Ultimately, runahead execution can prefetch data that data prefetchers miss, but overall the data prefetchers cover more long stalls and offer better performance.

VI. Conclusion and Member Contributions

Both Data Prefetching and runahead execution can reduce the impact of the memory wall, but in our studies we found that runahead execution alone has difficulty in mitigating these long latencies. Runahead, while theoretically offering perfect prefetching, is not able to provide much benefit on the chosen workloads due to the limited window in which it was able to execute. This limited window negatively affected the timeliness of runahead and lead to an overall speedup of only 2.3% in a standalone system. When combined with data prefetching, this benefit drops to 1.8% over the prefetcher-only versions. Overall, runahead execution did provide some benefit to both the baseline and data prefetching combinations, but the

Configuration	IPC Speedup Relative to Baseline
GHB	6.7%
Markov	0.9%
Nextline	0.1%
Stream	9.3%
Stride	5.6%
Stride PC	6.0%
GHB & Stride	9.5%
GHB & Stride PC	10.5%
Stream & GHB	12.4%
Stream & Stride	10.5%
Stream & Stride PC	11.2%
Runahead	2.3%
Runahead & GHB	8.7%
Runahead & Markov	2.8%
Runahead & Stream	10.3%
Runahead & Stride	6.0%
Runahead & Stride PC	8.7%
Runahead & GHB & Stride	13.4%
Runahead & GHB & Stride PC	12.9%
Runahead & Stream & GHB	11.5%
Runahead & Stream & Stride	12.1%
Runahead & Stream & Stride PC	13.0%

TABLE III
IPC SPEEDUP COMPARISON

benefit was minimal when compared to the cost associated with runahead execution.

In the future this work should take four directions:

- Run this on graph workloads. Graph workloads have extremely irregular memory access patterns that are inclined to the stalls that runahead needs for best performance.
- Use gem5 to get a full system simulation that covers both Virtual Memory and has a better cache model. We expect TLB misses to have long latencies and thus lead to more opportunities for runahead execution.
- Use multi-level data prefetching to see if you get a better result. Our experiments were limited to prefetchers in the L2 cache due to the Scarab simulator. Often caches have prefetchers at each level to improve throughput. Furthermore, we modeled a relatively small cache by modern standards and larger caches would involve higher latencies and possibly introduce more full window stalls.
- Compare Against Access-Execute Architectures.

A. Member Contributions

- Matthew: I created all parameters and configurations for testing and processed all results. Worked on paper and presentation.
- Karan: Implemented the ISB and IPCP prefetchers in scarab. Helped with debugging and worked on paper and presentation.
- Aditya: Created the docker infrastructure for NEOM NEOM fast running of scarab workloads. Helped with debugging, worked on paper, and presentation.
- Common: Debugging scarab, prefetcher configurations, and runahead execution bugs (A lot of this).

ACKNOWLEDGMENT

The authors would like to thank Dr. Calvin Lin, Quang Duong, and Chirag Sakhuja for a great semester. We learned a lot and really enjoyed the course! The authors would also like to thank Sanjana Yadav for providing some additional prefetcher configurations in scarab for our evaluation. Finally, the authors would like to thank Aniket Deshmukh for providing an implementation of precise runahead execution in scarab.

REFERENCES

[1] Milad Hashemi, Onur Mutlu, and Yale N. Patt. "Continuous runahead: Transparent hardware acceleration for memory intensive workloads". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783764.

[2] Sorin Iacobovici et al. "Effective Stream-Based and Execution-Based Data Prefetching". In: *Proceedings of the 18th Annual International Conference on Supercomputing*. ICS '04. Malo, France: Association for Computing Machinery, 2004, pp. 1–11. ISBN: 1581138393. DOI: 10.1145/1006209.1006211. URL: https://doi.org/10.1145/1006209.1006211.

[3] Akanksha Jain and Calvin Lin. "Linearizing irregular memory accesses for improved correlated prefetching". In: *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2013, pp. 247–259.

[4] Akanksha Jain and Calvin Lin. "Rethinking Belady's Algorithm to Accommodate Prefetching". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018, pp. 110–123. DOI: 10.1109/ISCA.2018.00020.

[5] D. Joseph and D. Grunwald. "Prefetching using Markov predictors". In: *IEEE Transactions on Computers* 48.2 (1999), pp. 121–133. DOI: 10.1109/12.752653.

[6] N.P. Jouppi. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers". In: *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*. 1990, pp. 364–373. DOI: 10.1109/ISCA.1990.134547.

[7] O. Mutlu et al. "Runahead execution: an alternative to very large instruction windows for out-of-order processors". In: *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* 2003, pp. 129–140. DOI: 10.1109/HPCA.2003.1183532.

[8] Ajeya Naithani et al. "Precise Runahead Execution". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 397–410. DOI: 10.1109/HPCA47549.2020.00040.

[9] K.J. Nesbit and J.E. Smith. "Data Cache Prefetching Using a Global History Buffer". In: *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. 2004, pp. 96–96. DOI: 10.1109/HPCA.2004.10030.

[10] Samuel Pakalapati and Biswabandan Panda. "Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching". In: *2020 ACM/IEEE 47th Annual International Symposium*

on Computer Architecture (ISCA). 2020, pp. 118–131. DOI: 10.1109/ISCA45697.2020.00021.

[11] *Scarab*. 2022. URL: https : / / github . com / hpsresearchgroup/scarab.

[12] Joel M. Tendler. et al. “Memory Prefetching Using Adaptive Stream Detection”. In: *2002 IBM Journal of Research and Development*. Vol. 46. 1. 2002, pp. 397–408. DOI: 10.1147/rd.461.0005.