Sieve: A Generalizable Perceptron Prefetch Filter

Slava Andrianov
University of Texas at Austin

Emily Guo
University of Texas at Austin

Annie Hu
University of Texas at Austin

***Abstract*—This paper introduces Sieve, a generalizable filtering approach towards prefetching that aims to learn and prevent mispredictions. We build on the previous work of Perceptron-Based Prefetch Filtering [2] and Sandbox Prefetching [9]. We first provide a generalizable implementation of the perceptron filter that can be applied with any underlying prefetcher. Then, we enable configuring our filter with multiple underlying prefetchers at once, placing an emphasis on simultaneous evaluation of multiple active prefetchers. Finally, we experiment with providing custom feature input to underlying prefetchers, and sandboxing, and integration between the prefetch filter and the cache replacement policy. Overall, our work seeks to tackle the fundamental trade-off of coverage and accuracy in prefetcher design.**

We describe our design and evaluate it in the last-level cache on 13 parallel application benchmarks of the GAP and PARSEC workload suites. In the single-prefetcher configuration, we apply Sieve's filtering mechanism to three prefetchers, and compared to a no-prefetching baseline, we improve performance by 1.73%-3.08%, which outperforms these prefetchers without filtering. In the multi-prefetcher configuration, we apply Sieve's filtering mechanisms to the same three prefetchers, and we find that Sieve again outperforms with a 3.39% speedup over no-prefetching. For a 4-core system, the multi-prefetching configuration of Sieve is able to improve IPC by 4.15%.

I. Introduction

Memory latency is a major factor limiting CPU performance. Previous prefetching work has attempted to mitigate this issue by learning patterns in memory accesses in order to predict and fetch the next demand access earlier to hide memory latency and improve program performance. However, these prefetching techniques are not 100% accurate, resulting in mispredicted prefetched lines occupying space in the cache and wasting memory bandwidth.

Previous work has focused on improving prefetching accuracy by learning heuristics such as correlations in temporal or spatial locality, or by recognizing common irregular access patterns. However, there are trade-offs between coverage and accuracy within prefetcher design, where a prefetcher can issue requests more aggressively with the aim of further reducing the number of future cache misses, but this can result in a reduction in accuracy due to the difficulty of predicting further into the future and the increased number of prefetched lines entering the cache.

The Perceptron-Based Prefetch Filter (PPF) [2] introduced a mechanism to reduce the effects of this trade-off by allowing an underlying prefetcher to aggressively suggest prefetches, and then using a perceptron to learn and decide on which prefetch requests should proceed to the cache hierarchy. The original PPF presents an implementation that is closely tied to the specific underlying prefetcher they evaluated. In this paper, we propose Sieve, which includes a generalizable implementation of PPF that can be applied to any prefetcher. We utilize default weights and features from information that is available to any prefetcher. In this single-prefetcher setting, we evaluate which features are the most useful and relevant when applied to any prefetcher. This also allows us to analyze how the filter performs differently when applied to different prefetchers in order to determine the impact of the filter in the overall decision-making process.

Furthermore, we enable Sieve to filter out prefetches from several active prefetchers at once. This intuitively allows us to benefit from each of the underlying prefetcher's unique contributions and increases our adaptability to a wide range of workloads. In the multi-prefetcher setting, we experiment with a single shared filter as well as one filter per prefetcher to determine the effects of helpful cross-learning or potential interference, given the storage trade-offs. We also experiment with incorporating a sandboxing [9] approach into the multi-prefetcher setting in order to determine how filtering improves upon sandboxing in general.

To match the original custom-tailoring aspects of the PPF implementation as described in the paper [2], Sieve allows for an expanded custom feature input so underlying prefetchers can still offer their own relevant characteristics to the filter.

Finally, we experiment with Sieve working in conjunction with a replacement policy, using Sieve's confidence as feedback for the replacement policy in order to guide insertion decisions.

To summarize, this paper makes these main contributions:

- Generalize the perceptron-based prefetch filter to any singular underlying prefetcher as well as multiple underlying prefetchers, in which case we can target several complex access patterns at once.
- Introduce a novel combination of filtering and sandboxing simultaneously, which outperforms sandboxing by itself.
- Allow for a single shared filter between multiple prefetchers, or a unique filter per prefetcher.
- Perform multi-class cache insertion prediction by integrating the confidence of the filter with the RRIP [4] values of the SHiP replacement policy.

II. Motivation and Related Work

In this section, we will discuss related work in the prefetching field and the two previous work that we primarily build on to create Sieve. In Section 3, we will elaborate on how the following work is incorporated into the implementation and testing of Sieve. By first describing existing temporal and

spatial prefetching approaches, we will motivate why optimizing the trade-off between coverage and accuracy through sandboxing and filtering is necessary, and hence the motivation behind Sieve's unique combination of these two aggression-tuning techniques.

A. Temporal Prefetchers

Temporal prefetching focuses on predicting what data will be accessed after other data based on a sequence of events regardless of how close the data are in memory. Often, this comes in the form of correlating addresses and identifying temporal streams. One example of a temporal prefetcher is the Markov prefetcher, which transforms an address stream into a Markov model in order to predict what reference will follow a given reference [5].

B. Spatial Prefetchers

Spatial prefetching takes a different approach by instead considering the spatial locality of accesses. For example, a next-line prefetcher is a spatial prefetcher because when it sees address A, it will prefetch address A + 1. This can be generalized to an offset or stride prefetcher, which will learn an offset/stride of Δ, so when it sees address A, it will prefetch A + Δ. Often, a stride prefetcher will only issue a request if it is somewhat confident that it will be useful, for example it may wait to see address A and address A + Δ before requesting A + 2 * Δ.

The Best Offset Prefetcher (BOP) is a stride prefetcher that has a predefined list of offsets and then selects a singular best offset from this list at run-time [8]. This prefetcher maintains a table of the trigger address of recent requests, and then searches through these addresses to determine if the currently requested line would have been timely. In this case, an offset's corresponding score is incremented, and at the end of a learning period, the offset with the highest score is selected as the current prefetch offset.

The Signature Path Prefetcher (SPP) is another spatial prefetcher that predicts the next delta, using a 12-bit compressed signature of the previous 4 deltas to store history. SPP extends this history to include the newly predicted delta in order to generate a new signature that can be used to make another prediction. This is part of SPP's lookahead mechanism, which allows SPP to deeply speculate. However, to avoid losing accuracy for coverage, SPP also introduces a confidence metric that is used to throttle the prefetching depth dynamically.

C. Irregular Prefetchers

Up until now, we have discussed regular access patterns that follow a heuristic such as temporal or spatial locality. However, programs often exhibit irregular access patterns that cannot be defined by these simple heuristics. In this case, a prior work has proposed the Indirect Memory Prefetcher (IMP), which aims to learn a specific, but common irregular access pattern [11].

IMP targets indirect memory accesses of the form A[B[i]], which exhibit little spatial locality but are common in graph applications for accessing neighbor vertices. IMP captures this indirect pattern by reading in advance the contents of B[i], as that is the unpredictable part of this pattern, and then detecting which accesses are reads to an array so it can compute the start of the array and the size of an element in the array to predict future indirect accesses.

In addition to these prefetching approaches, there are additional techniques such as sandboxing and filtering that aim to improve prefetching performance.

D. Underlying Concept: Perceptron Prefetch Filter

All of the prefetchers we have described thus far suffer from an inherent trade-off between coverage and accuracy. A Perceptron-Based Prefetch Filter (PPF) [2] attempts to mitigate this trade-off by allowing an underlying prefetcher to be more aggressive and hence have higher coverage while a perceptron filter approves or rejects the underlying prefetcher's suggested prefetches to avoid negatively impacting accuracy. The filter considers several features of the prefetch request, and based on the output confidence of the perceptron it will decide if the prefetch request should actually be issued, and if so, whether the prefetched line should go to the L2 cache or the LLC depending on two confidence thresholds.

While PPF is theoretically applicable to any underlying prefetcher, the original paper focused on a case-study with SPP as the underlying prefetcher, and used features specific to SPP, such as the lookahead depth, signature, and confidence counter on top of tuning SPP to be more aggressive by discarding its internal throttling mechanism. However, the core infrastructure of PPF such as its weight tables, prefetch table, reject table, and ability to prefetch into L2, LLC, or reject a suggested prefetch still comprises an effective system that could be modularized and adapted to a variety of prefetchers.

E. Underlying Concept: Sandbox

Program behaviors vary significantly across and within workloads, so it is natural there is not a single prefetcher that is ideal in every situation. All prefetchers are forced to make a coverage and accuracy trade-off, especially because many prefetchers aim to target a specific access pattern. Existing prefetchers attempt to address this issue by integrating adaptive characteristics to help the prefetcher dynamically alter its prefetching behavior in conjunction with the program evolution.

However, another approach to this problem is the concept of sandboxing, originally introduced in Sandbox Prefetching [9]. This paper proposes evaluating multiple candidate prefetchers round-robin style such that in every evaluation period, only the best-performing prefetcher is allowed to issue requests to main memory once its performance has surpassed a certain threshold. In the original paper, the concept of sandboxing was applied to sixteen candidate stride prefetchers with varying offsets.

In the past, we have extended upon the original paper's sandboxing approach with our ++Sandbox design. In ++Sandbox, we use advanced candidate prefetchers rather than simple

stride prefetchers, as well as an option of no-prefetching if none of the candidates are performing above a certain threshold. Additionally, in ++Sandbox, we evaluate all of the candidate prefetchers simultaneously rather than in a round-robin fashion as the original Sandbox proposes.

The central idea behind both sandboxing designs is to continuously evaluate what the best prefetcher for the current program phase is. However, both suffer from the same flaw where during a phase change, the outgoing prefetcher may still issue requests, hence polluting the cache and wasting memory bandwidth until a new best prefetcher is confirmed.

III. Solution: Sieve

First, we introduce the high-level overview of Sieve, a generalized PPF, and then we discuss the implementation-specific details.

A. High-Level Overview

1) Generalized PPF: Sieve strips PPF down to its core infrastructure that is applicable to all prefetchers. For example, all prefetchers have access to program context features useful for making predictions such as delta sequences, page addresses, cache line offsets, and in some cases, the program counter. Thus, these program context features are either used directly or combined to generate features that our perceptron uses for training and prediction. This is the key behind generalizing PPF for application to any prefetcher.

However, as the case-study in the original PPF paper motivates, the filter may benefit from additional computed data and additional features available from the underlying prefetcher. Thus, Sieve allows additional custom feature input that is easily configured to provide the prefetch filter with additional features. These features will have corresponding weights which are updated and factored into the filter's decisions.

We describe the full design and implementation of the custom feature input implementation in Section IV-3.

2) Multi-Prefetcher Configuration: Furthermore, Sieve supports single and multi-prefetcher configurations. For the multi-prefetcher configuration, we support sharing a single filter across all prefetchers, or one filter per prefetcher and experimentally determine which design performs better.

The default multi-prefetcher configuration simply manages all of the underlying prefetchers, allowing any of them to suggest prefetches which are then filtered accordingly. However, we also experimented with a sandboxed version of Sieve to allow for a novel, unique combination of filtering and sandboxing.

The sandboxed multi-prefetcher configuration is conceptually very similar to sandboxing. However, the key advantage that Sieve gains from both filtering and sandboxing compared to only sandboxing is the ability to intelligently and dynamically filter out the suggested prefetches suggested by the best-performing prefetcher, rather than blindly issuing any prefetch suggested by the best-performing prefetcher once its score has surpassed a certain threshold. This helps to avoid wasting memory bandwidth and polluting the cache.

3) Underlying Prefetchers: We chose to evaluate three advanced candidate prefetchers - SPP, IMP, and BOP - as Sieve's underlying prefetchers. SPP was evaluated in the original paper, so it serves as a baseline of comparison for Sieve's generalized PPF against the original paper's PPF that was custom-tailored to SPP. Furthermore, we wanted to extend our evaluation to irregular access patterns with IMP as one of the main benefits of activating multiple prefetchers at once is that we can capture a variety of patterns simultaneously, enjoying the benefits each new underlying prefetcher brings. Lastly, we chose BOP because it is the winner of DPC-2, and its goal of dynamically finding the best offset is reminiscent of the original sandbox prefetcher.

B. Sieve Implementation

Sieve's Generalizable PPF implementation uses the core infrastructure described in the original PPF paper [2]. Each feature has a corresponding weight table of 1,024 entries mapping feature values to weights, where each entry is a 5-bit saturating counters ranging from -16 to +15. Additionally, we retain the 1,024-entry Prefetch Table storing accepted prefetches and the 1,024-entry Reject Table storing rejected prefetches, both of which use an LRU replacement policy. The Prefetch and Reject Table both map the physical address of a line to the feature values needed to re-index into the weight tables for training.

1) Features Chosen: The key difference between Sieve's Generalizable PPF and the original PPF design is the specific features we use that are agnostic of the underlying prefetcher. We place an emphasis on incorporating features that are easily available from the program context at the LLC level and useful to most prefetcher designs. Many of our features are inspired by the features used in the original PPF implementation and the features explored in the creation of the Pythia prefetcher [1].

However, the recent work striving to "kill the program counter" (due to the practical difficulty of making the program counter accessible to the cache hierarchy) has influenced us to make our design flexible such that Sieve can be easily configured to be used with or without program counter features [6].

We tested additional features such as the cache line and last four page offsets, but we experimentally found that they were only a small factor in the decision-making process which led us to remove them. The final list below is the pruned list of features that we found to be useful and significant in the perceptron filter's decisions.

Program Counter Independent Features

- **Physical Offset**
- **Physical Page Number**
- **Load Address Delta**
- **Last Four Load Deltas**
- **Physical Offset XOR Load Address Delta**

Program Counter Based Features

- **Program Counter (PC)**
- PC **XOR** $PC_1 >> 1$ **XOR** $PC_2 >> 2$

- **PC XOR Delta**

2) Thresholds Chosen: The inference process is the same as that of PPF. When a request is made, each feature indexes into a table to retrieve a weight, and then all of the weights are summed. If the sum is below a low threshold, then the prefetch is rejected. If the sum is between the low and high threshold, then the prefetch is sent to the LLC, otherwise, it is sent to the L2 cache as we have high confidence in this prefetch's usefulness. After some experimental testing, we found that a high threshold of 90 and a low threshold of 25 to be optimal, matching the values used in the original PPF design.

Similarly, we mirror the training process in the original PPF paper. Training is done on the L2 access stream, and we also maintain a high and low training threshold to avoid over-saturation of the weights while maintaining the training speed. Through experimental testing, we chose 80 and -80 for the high and low training thresholds respectively.

3) gem5 Specific Implementation Details: We simulate Sieve in the gem5 simulator. To make Sieve applicable to any underlying prefetcher in gem5 specifically, we first made Sieve and Sieve's filter() method accessible to all prefetchers. Then, to apply Sieve to a prefetcher, the only modification that needs to be made is to simply call filter() whenever the underlying prefetcher wishes to issue a prefetch to the main memory. Inside Sieve's filter() method, Sieve will determine if the suggested prefetch should be accepted, and only then will Sieve actually allow the prefetch to be sent out. Thus, in terms of the software changes required, Sieve is very convenient and accessible.

IV. Additional Experimentation

In this section, we will describe our additional experiments and the relevant modifications made to support these experiments.

1) Sandboxed Multi-Prefetcher: On top of the default multi-prefetcher configuration, we support a sandboxed version where we filter out the prefetches suggested by the best-performing prefetcher. The motivation behind this experiment is that sandboxing by itself will allow all prefetches to go through once the best-performing prefetcher is activated, regardless of whether or not its suggested prefetches are useful, leading to wasted resources. Thus, we aim to improve upon sandboxing by itself, and intuitively, by applying Sieve's filter on top of sandboxing, we are able to intelligently filter out the suggested prefetches from the best prefetcher, a major improvement over only using a static score threshold.

In terms of the implementation, we essentially created a sandbox evaluating all of the underlying prefetchers, and then once the best-performing prefetcher is determined and its performance surpasses the score threshold, it is allowed to suggest prefetches. At this point, Sieve's filter is applied to filter out suggested prefetches.

2) Shared Filter vs. Individual Filters: In this experiment, we aimed to determine if a single shared filter across multiple prefetchers or an individual filter per prefetcher would perform better. The motivation behind using individual filters despite the higher storage cost is that it may reduce aliasing between the prefetchers potentially accessing the same table entries and interfering with each other.

The main modification we had to make was to create a filter associated with each underlying prefetcher. Thus, instead of using one globally accessible Sieve for all prefetchers, we created a Sieve that was only accessible to a specific prefetcher, which that prefetcher would use for filtering its prefetches.

3) Expanded Custom Feature Set: In this experiment, we aimed to allow the underlying prefetchers to offer relevant metadata as custom features to Sieve. As discussed earlier, the motivation behind this experiment is that the filter may benefit from this additional metadata to help it determine if a prefetch is useful or not.

In order to support an expanded, custom feature set, we had to add additional data structures. We decided to limit the number of additional features to 3 because the original PPF paper only added 3 features specific to SPP and we were only planning on testing up to 3, but we should note that our implementation is simple to scale up to more features as necessary.

Thus, for the 3 additional features, we had to add 3 additional weight tables, 3 supplemental prefetch tables, and 3 supplemental reject tables. The core behavior of Sieve remains the same, however, with each prefetch request, the underlying prefetcher must also send values for the additional features so they can be stored in the relevant additional weight tables. Furthermore, depending on whether the prefetch is accepted or rejected, the feature values must be placed in the corresponding prefetch or reject table such that when training occurs via the standard Prefetch and Reject table, the supplemental prefetch and reject tables will also be accessed to adjust the weights for the custom features in the additional weight tables.

If a prefetcher does not wish to use additional custom features, then these tables can be easily deactivated.

4) SHiP Integration: Several cache replacement policies attempt to predict the re-reference interval of blocks in the cache in order to determine their eviction priority [10]. However, many of these policies end up making predictions that are effectively binary. They choose between two possible RRIP values (where RRIP refers to Re-reference Interval Prediction), to assign to a cache line upon insertion, and then internally manipulate and update this value as lines within the cache are referenced. For example, the SHiP replacement policy uses the insertion logic of "if (SHCT[Signature] == 0) 3; else 2;" and for promotion, it always sets the RRIP value to 0. For SHiP, a smaller RRIP value indicates a near re-reference interval, which larger RRIP values indicate more distant re-reference intervals.

As such, we have identified an opportunity to use Sieve in conjunction with the SHiP replacement policy in order to more intelligently assign RRIP values. We propose transforming the outputted confidence value of the perceptron filter into an RRIP value. The intuition behind this idea is that a higher

confidence value means that we will insert into the L2 cache over the LLC, implying that when the demand access occurs for the same line, it will first search the L2 and find the line rather than the LLC. Thus, we prefer this line having a smaller re-reference interval, so we can assign it a 0 or 1 RRIP value rather than a 2 or 3, which is the default for SHiP.

As for the implementation, we have an additional table which is indexed into using the prefetch address to access the confidence value (the sum of the weights) of the filter when it was evaluating that prefetch. Then, if the confidence is above 80 or 70, then we predict a RRIP value of 0 or 1 respectively.

TABLE I
MEMORY HIERARCHY

Cache Level	Specifications
L1I	32 KB, 8 ways
L1D	32 KB, 8 ways
L2	1 MB, 16 ways
LLC	8 MB, 16 ways

V. METHODOLOGY

As mentioned earlier, we follow the original PPF design in training all of Sieve's underlying prefetchers and the prefetch filter itself on the L2 access stream, and we insert prefetches into both the L2 and LLC cache. The LRU replacement policy is used on all levels of the cache, except for in the SHiP integration experiment where the SHiP replacement policy is applied to the LLC.

Simulator: We use the execution-driven simulator gem5. Gem5 models a complex out of order CPU and memory hierarchy, and is an "execute-in-execute" simulator, allowing for wrong-path execution. We used gem5 in full-system mode, simulating the interactions of the entire system and OS kernel for a more realistic simulation. Table 1 details the memory hierarchy configuration. We did not compare Sieve to the original PPF paper's results as they used a different simulation technology called ChampSim that is less detailed than gem5 (for example, it does not support wrong path execution and it models each instruction with a fixed latency), and therefore would not serve as a fair baseline. Furthermore, there is no public gem5 implementation of the original PPF paper available to us for comparison.

Benchmarks: We perform analysis on 13 parallel-application benchmarks from the GAP and PARSEC suites, targeting irregular graph codes and scientific workloads.

Metrics: In our results, we report the percent IPC speedup over a no-prefetching baseline. Additionally, we provide the geometric mean of the percent IPC speedup when applicable to show the overall mean speedup over no-prefetching.

VI. RESULTS

All results are single-core and normalized to no-prefetching unless otherwise mentioned. For our naming convention in our results, **Sieve-S** indicates Sieve is applied in a single-prefetcher setting and **Sieve-M** indicates Sieve is applied in a multi-prefetcher setting. For the multi-prefetcher setting, we always test SPP, IMP, and BOP as the underlying prefetchers, so when we refer to **Sieve-M-ALL**, this refers to this configuration for brevity.

A. Single Prefetcher Comparisons

We first present our results for **Sieve-S**.

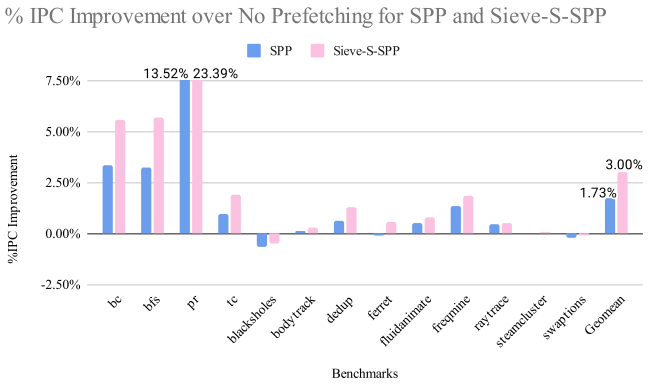

Fig. 1. SPP vs. Sieve-S-SPP

In Figure 1, we compare **Sieve-S-SPP** and SPP normalized to no-prefetching, and we show a 3.00% IPC improvement on average as measured by the geometric mean compared to SPP's 1.73%. We improve on every benchmark, even on benchmarks where both SPP and **Sieve-S-SPP** perform worse than no-prefetching such as blackscholes.

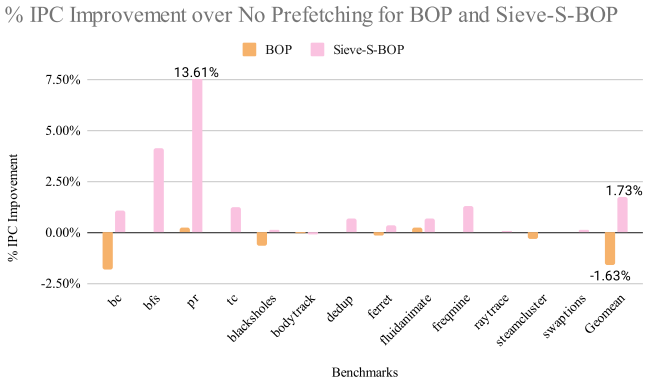

Fig. 2. BOP vs. Sieve-S-BOP

In Figure 2, we compare **Sieve-S-BOP** and BOP. We show a 1.73% IPC improvement compared to BOP, which performs 1.63% worse than no-prefetching on these benchmarks on average. This demonstrates that BOP may be aggressively issuing prefetches if an offset's score is high enough and a prefetch is predicted to be timely, which may not take into account enough factors to produce useful prefetches. Thus, this is a prime example of when we are able to intelligently filter out aggressive prefetches using the filter's overall expanded view of the program context.

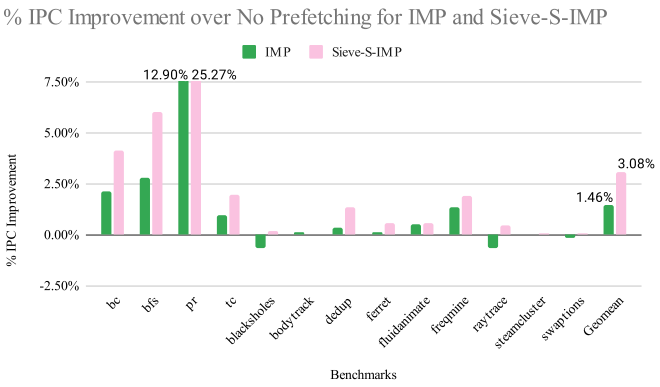

Fig. 3. IMP vs. Sieve-S-IMP

In Figure 3, we compare **Sieve-S-IMP** and IMP. We show a 3.08% IPC improvement compared to IMP, which shows a 1.46% improvement over no-prefetching. This is our greatest speedup over no-prefetching. This result makes sense as the benchmarks from the GAP suite exhibit irregular graph codes, and IMP should perform well on graph applications by specifically learning the indirect memory access pattern of A[B[i]] which is often used in graph-related algorithms to find neighboring vertices. However, we note that on the non-GAP workloads, our filter still improves performance as we improve on all benchmarks except for bodytrack, where we show a negligible decrease in performance of 0.11%.

Overall, in the single-prefetcher setting, we show that our dynamic filtering of useless prefetches consistently improves performance.

B. Multi Prefetcher Comparisons

Now, we present our results for **Sieve-M** in both single-core and 4-core evaluations. For our naming convention, SF indicates a single shared filter among prefetchers, IF indicates individual filters (one filter per prefetcher), SB indicates sandboxed, and NSB indicates not sandboxed.

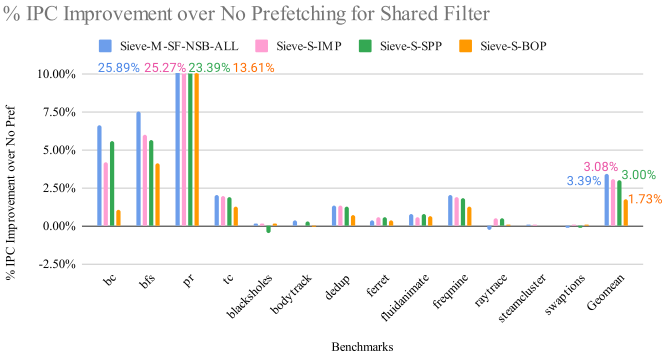

Fig. 4. Sieve applied to SPP, BOP, and IMP with a Single Shared Filter

Our single-core multi-prefetcher results are shown in Figure 4. In this configuration, Sieve shares a single filter among all of the underlying prefetchers. Here, Sieve is able to achieve a mean speedup of 3.39%, which performs better than when Sieve is applied to any of the prefetchers by themselves. This result demonstrates that Sieve benefits from more than one underlying prefetcher as it is able to enjoy the benefits of each prefetcher's unique contributions simultaneously.

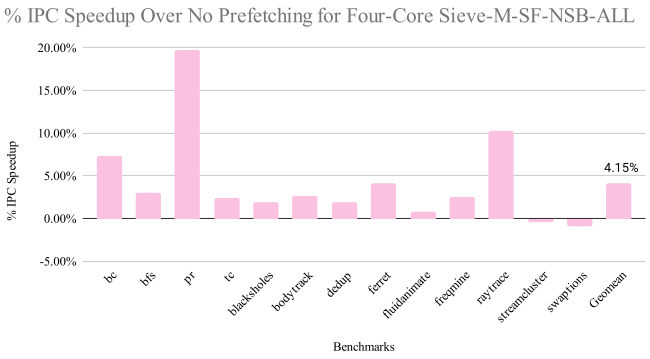

Fig. 5. 4-Core Results for Sieve's Single Shared Filter

Our 4-core results for the multi-prefetcher configuration with a single shared filter are shown in Figure 5. Once again, we show improvement on every benchmark with a geometric mean speedup of 4.15%.

C. Sandboxing Experiment Analysis

Now, we present our findings of experimenting with our unique combination of sandboxing and filtering compared to only sandboxing. With regards to our naming convention, these results all use a single shared filter, so we refer to this configuration as **Sieve-M-SF-SB-ALL**. We show the results of sandboxing with individual filters in Section VI-D.

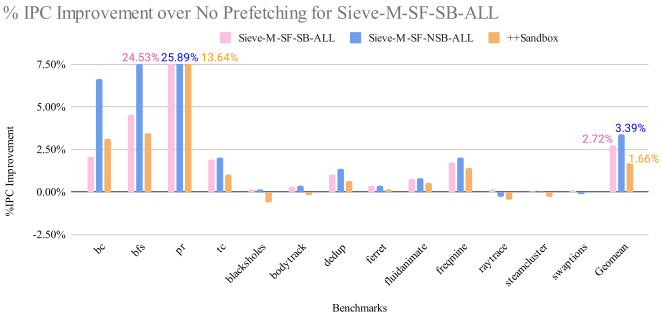

Fig. 6. Sieve's Single Shared Filter Results with Sandboxing

We find that **Sieve-M-SF-SB-ALL**'s IPC improvement of 2.72% outperforms ++Sandbox which only shows an IPC improvement of 1.73%. Thus, our unique combination of both filtering and sandboxing outperforms only sandboxing, and this confirms our hypothesis that the filtering mechanism can prevent useless prefetches from the best-performing prefetcher. This demonstrates that accepting any prefetch from the best-

performing prefetcher once its score has surpassed a threshold is not enough, and a filter can further improve performance.

However, our (geometric) mean speedup of 2.72% is still lower than 3.39% speedup without sandboxing. This is likely because in the sandboxing approach, only the best-performing prefetcher can suggest prefetches, whereas, without sandboxing, any of the underlying prefetchers can suggest prefetches.

Intuitively, if any of the underlying prefetchers can suggest prefetches, then we can capture more than one type of access pattern at once. For example, suppose we have a spatial prefetcher and an irregular access prefetcher as our underlying prefetchers. In that case, theoretically, we are able to predict spatial and irregular access patterns simultaneously, in contrast to sandboxing, where we are limited to the pattern of the best-performing prefetcher targets. This benefit is especially strong when program phase changes occur because in the non-sandboxed version of Sieve, the best prefetcher for the incoming phase can have its prefetches be issued and contribute to improving program performance right away, instead of having to endure the transition between active prefetchers like in a sandboxed environment.

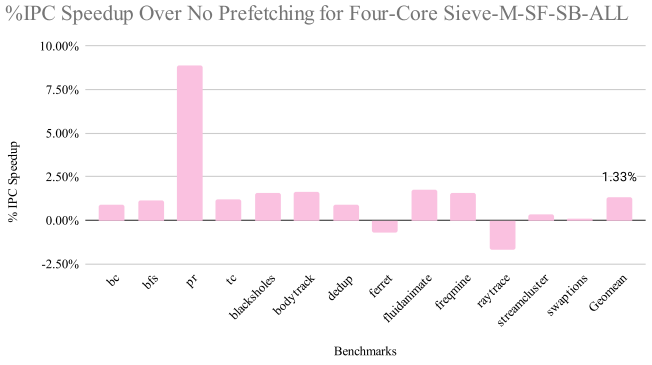

Fig. 7. 4-Core Results for Sieve's Single Shared Filter with Sandboxing

In Figure 7, we present our 4-core results for **Sieve-M-SF-SB-ALL**, which do not perform well with a marginal improvement of 1.33% speedup overall compared to the 4-core results for **Sieve-M-SF-NSB-ALL**, which attain a 4.15% speedup. First, we suspect that the sandboxed version performs worse than the non-sandboxed version for the reasons explained above for the single-core results. This problem of only considering prefetches from the best-performing prefetcher is likely exacerbated in a 4-core simulation because the best prefetcher is likely not the same across cores.

Second, we suspect that our 4-core results may not be achieving their full potential for both the sandboxed and non-sandboxed versions. Thus, **Sieve-M-SF-NSB-ALL**'s results may also fall short of their true potential, but performance still improved due to the benefits of filtering. However, for the sandboxed version, any performance gain from filtering was still not enough to hide the key weakness of ++Sandbox in multi-core environments. It is likely that not all four cores will have the same best prefetcher for their current task, so it is likely that some of the cores are running a suboptimal prefetcher while also causing aliasing within the sandbox prefetcher selection. We discuss the potential of remedying this problem with a one-filter-per-core solution as future work in Section VII-B.

D. Shared vs. Individual Filters Experiment Analysis

In this section, we present our findings for our experiment comparing a single shared filter among multiple prefetchers and an individual filter per prefetcher. In Figure 8, we compare the non-sandboxed Sieve filter configurations, and in Figure 9, we compare the corresponding sandboxed Sieve configurations.

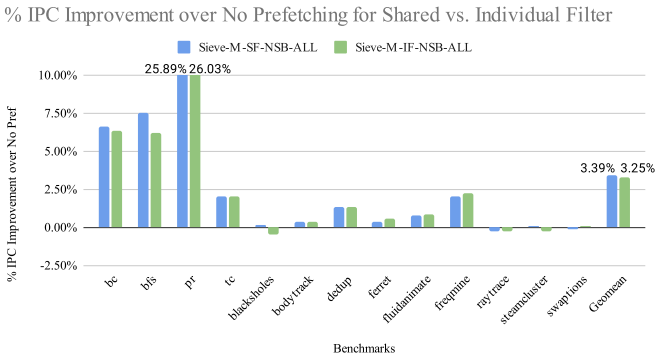

Fig. 8. Shared Filter vs. Individual Filter

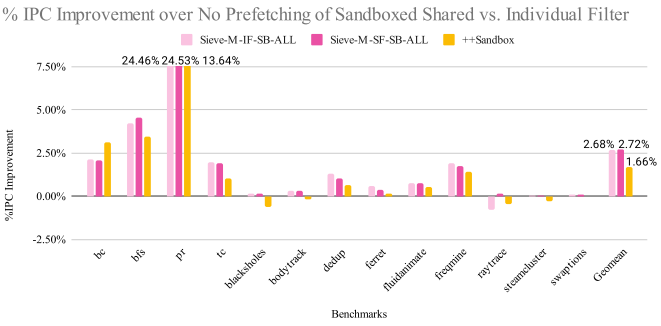

Fig. 9. Sandboxed: Shared Filter vs. Individual Filter

Regardless of sandboxing, the performance is very similar. However, the shared filter configuration overall performs better than the individual filter configuration with an IPC speedup of 3.39% compared to 3.25% for non-sandboxed Sieve, and a speedup of 2.72% compared to 2.68% for sandboxed Sieve. This intuitively makes sense as regardless of the filter configuration, a good or bad prefetch should be identifiable by any filter.

Sieve with multiple active prefetchers tends to perform especially well on graph based benchmarks such as bc, pr, and bfs regardless of the configuration, likely due to the ability of the component IMP prefetcher to excel at this

task. Meanwhile, lower performance is seen on raytrace and streamcluster, but these tend to be weaker benchmarks for the underlying prefetchers as well so it is unlikely that a filtered version would see much of an improvement.

We observe that on benchmarks with repetitive actions suited to a specific prefetcher, the individual filter configuration performs better. However, when a benchmark alternates behavior, a single shared filter produces the best results.

Under an individual filter configuration, feedback from training is only applied to the filter of the prefetcher which requested the line, leading to slower training than a shared prefetcher design as some underlying prefetchers will receive less training than others. While this can prevent aliasing, the slower training proves to be more detrimental in cases where the optimal prefetching policy may vary over the program phases. We observe that it is more beneficial to use a single shared filter which is always trained and ready to review suggested prefetches, rather than trying to switch prefetchers and then needing to warm up that prefetcher's prefetch filter to the current program state.

E. Custom Feature Input Experiment Analysis

In this section, we demonstrate our findings for a custom-tailored version of Sieve applied to SPP, which we refer to as **Sieve-S-SPP-Custom**.

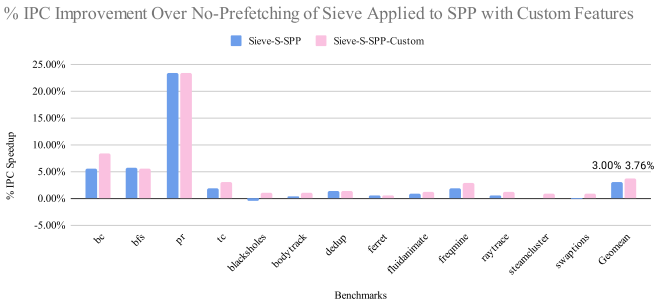

Fig. 10. Sieve applied to SPP with a Custom Expanded Feature Set

In this experiment, we expanded SPP by including its Confidence, Signature, and Lookahead Depth as additional custom features, matching the original additional input provided in the original PPF paper [2]. We find that the custom input results in a speedup of 3.76% compared to 3.00% for **Sieve-S-SPP** without custom input.

This performance improvement confirms our hypothesis that providing the filter with additional available metadata that the prefetcher uses to make prediction decisions can help provide the filter with an expanded view of the overall program and prefetcher interactions. Although Sieve is generalized to any prefetcher, it can easily be adjusted for additional input that improves the filter's approval decisions and therefore the overall performance.

F. SHIP RRPV Analysis

We refer to the version of Sieve working in conjunction with SHiP applied to SPP as **Sieve-S-SPP-SHiP**. We compare this to a baseline of SPP by itself, as well as a baseline of **Sieve-SPP** running with SHiP as the replacement policy, which is referred to as "**Sieve-SPP** with SHiP" in Figure 11.

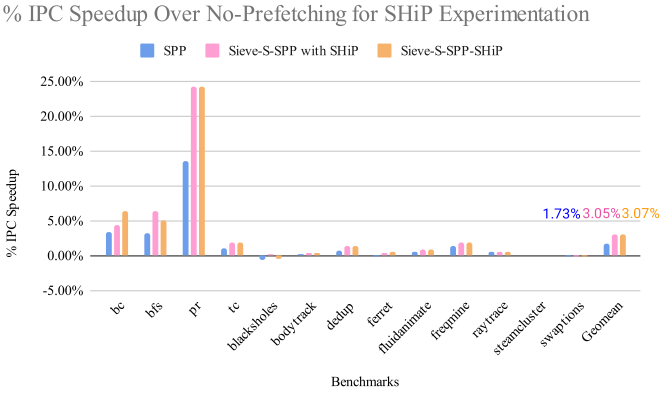

Fig. 11. SHiP Integration

In Figure 11, we observe a marginal improvement of 3.07% for **Sieve-S-SPP-SHiP** compared to 3.05% from running **Sieve-S-SPP** with SHiP as the replacement policy.

One reason we only improve marginally could be because we only alter the insertion RRIP value in the LLC, but many of our prefetches are issued to the L2 cache. In future work, we may concretely determine the ratio of prefetches issued to the L2 and LLC, but for now, we estimate that the number of prefetches issued to the LLC is low, so there was not a large room for improvement in the LLC to begin with.

Furthermore, we suspect that while we were able to improve, the correlation between the filter's confidence for inserting in the L2 or LLC and a near-immediate re-reference interval of the prefetched line may be lower than we thought, so the improvement was less drastic than expected. In future work, we may experiment with assigning more distant re-reference intervals, as we have only experimented with assigning more near intervals thus far.

VII. Future Work

A. Integrating Sandbox Score

Each prefetcher in the sandbox has an associated score corresponding to its number of useful prefetches. We plan on integrating this score as feedback to the perceptron filter for training purposes. The intuition behind this idea is that a prefetcher in the sandbox with a high score must have issued many useful prefetches, so then the filter should be more confident in that particular prefetcher. One concern with this approach is that it may result in "double-training" the filter, or perhaps introducing a confirmation bias if implemented incorrectly.

B. Custom Tailored Sieve-BOP

Similar to the custom-tailored PPF-SPP implementation, in the future we would like to experiment with a custom-tailored Sieve-BOP implementation. We are considering tuning the

underlying BOP prefetcher to be more aggressive by tuning its score reward mechanism, testing more offsets including negative offsets, and lowering or eliminating the BADSCORE threshold that prevents prefetching.

C. One Filter Per Core

Currently, our filter is not core-aware and this is reflected in our multi-core results. We plan to experiment with an implementation where we implement one filter per core so that the filter weights do not suffer from aliasing from different processes in different cores accessing and updating the same weights.

VIII. Conclusion

The Sieve prefetch filter shows significant promise in improving coverage without sacrificing accuracy, building on the previous work of the perceptron-based prefetch filter. In this paper, we provide the first generalized implementation of a prefetch filter that can be applied to a single underlying prefetcher or multiple underlying prefetchers at once.

Our generalized Sieve does not sacrifice customizability. We allow underlying prefetchers to easily and conveniently offer their own metadata to the filter as additional custom features, and show that this improves the filter's performance when applied to SPP, mirroring the case-study from the original perceptron-based prefetch filter work.

In particular, we show that multiple underlying prefetchers with a single shared filter is the optimal configuration as it enjoys each underlying prefetcher's unique contributions, allowing us to target several complex access patterns at once. This is in contrast to sandboxing approaches, which are limited to the prefetches of the best-performing prefetcher only. We also demonstrate that our unique and novel combination of filtering and sandboxing outperforms only sandboxing, and that filtering is a superior approach to the static score threshold currently used in sandbox prefetching.

Finally, we explore using our filter in conjunction with a cache replacement policy to guide insertion decisions for multi-class eviction priorities.

Sieve outperforms all of its underlying prefetchers in single- and multi-prefetcher settings. Sieve achieves a speedup of up to 3.08% in a single-prefetcher setting and a speedup of 3.39% in a multi-prefetcher setting, over no-prefetching. Furthermore, in a 4-core multi-prefetcher setting, Sieve achieves a speedup of 4.15% over no-prefetching.

Acknowledgment

Thank you to Christopher Hill and Lukas Zenick for their encouragement.

References

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

[2] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz and D. A. Jiménez, "Perceptron-Based Prefetch Filtering," 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), Phoenix, AZ, USA, 2019, pp. 1-13.

[3] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Davis, CA, USA, 2013, pp. 247-259.

[4] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10). Association for Computing Machinery, New York, NY, USA, 60–71

[5] Doug Joseph and Dirk Grunwald. 1997. Prefetching using Markov predictors. In Proceedings of the 24th annual international symposium on Computer architecture (ISCA '97). Association for Computing Machinery, New York, NY, USA, 252–263. https://doi.org/10.1145/264107.264207

[6] Jinchun Kim, Elvira Teran, Paul V. Gratz, Daniel A. Jiménez, Seth H. Pugsley, and Chris Wilkerson. 2017. Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17). Association for Computing Machinery, New York, NY, USA, 737–749. https://doi.org/10.1145/3037697.3037701

[7] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson and Z. Chishti, "Path confidence based lookahead prefetching," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 2016, pp. 1-12, doi: 10.1109/MICRO.2016.7783763.

[8] P. Michaud, "Best-offset hardware prefetching," 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona, Spain, 2016, pp. 469-480, doi: 10.1109/HPCA.2016.7446087.

[9] S. H. Pugsley et al., "Sandbox Prefetching: Safe run-time evaluation of aggressive prefetchers," 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, USA, 2014, pp. 626-637, doi: 10.1109/HPCA.2014.6835971.

[10] C. -J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely and J. Emer, "SHiP: Signature-based Hit Predictor for high performance caching," 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, 2011, pp. 430-441.

[11] X. Yu, C. J. Hughes, N. Satish and S. Devadas, "IMP: Indirect memory prefetcher," 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Waikiki, HI, USA, 2015, pp. 178-190, doi: 10.1145/2830772.2830807.