PARS: A Pattern-Aware Spatial Data Prefetcher Supporting Multiple Region Sizes

Yiquan Lin, *Graduate Student Member, IEEE*, Wenhai Lin, Jiexiong Xu, *Graduate Student Member, IEEE*, Yiquan Chen, Zhen Jin, Jingchang Qin, Jiahao He, Shishun Cai, Yuzhong Zhang, Zonghui Wang, and Wenzhi Chen

***Abstract*—Hardware data prefetching is a well-studied technique to bridge the processor-memory performance gap. Bit-pattern-based prefetchers are one of the most promising spatial data prefetchers that achieve substantial performance gains. In bit-pattern-based prefetchers, the region size is a crucial parameter, which denotes the memory size that can be recorded by a pattern or prefetched by a prediction. However, existing bit-pattern-based prefetchers only support one fixed region size. Our experiment shows that the fixed region size cannot meet the requirements for numerous applications and leads to suboptimal performance and high hardware overhead. In this article, we propose PARS, a pattern-aware spatial data prefetcher supporting multiple region sizes. The key idea of PARS is that it supports multiple region sizes, enabling it to simultaneously enhance application performance while reducing the hardware overhead. Moreover, PARS supports dynamically switching appropriate region sizes for different patterns through an adaptive RS-switching mechanism. We evaluated PARS on numerous workloads and results show that PARS provides an average performance improvement of 40.6% over a baseline with no data prefetchers and outperforms the two state-of-the-art prefetchers Bingo by 2.1% (up to 24.4%) and Pythia by 3.9% (up to 111.2%) in the single-core system. In the four-core system, PARS outperforms Bingo by 5.0% (up to 66.0%) and Pythia by 5.4% (up to 177.9%).**

***Index Terms*—Cache, data prefetching, hardware prefetching, microarchitecture.**

I. Introduction

FREQUENT cache misses induce significant latency, severely constraining the processor performance. The processor-memory gap affects not only general-purpose processors but also embedded the system processors. As embedded system processors are increasingly expected to perform tasks with high-performance demands [40], [41], [45], such as those in autonomous driving and industrial automation [6], [8], [11], optimizing memory access latency and enhancing performance have become crucial.

Hardware data prefetching is a widely adopted technique to bridge the processor-memory performance gap. It predicts future memory addresses to be accessed and preloads the data into the on-chip cache. Existing prefetching techniques primarily target general-purpose processors, effectively minimizing the number of cache misses and enhancing performance. However, these techniques often incur substantial area costs and increase memory traffic [15], [25], [38], [42], leading to higher power consumption [42], [48], which makes them less suitable for deployment in the embedded systems.

Bit-pattern-based prefetchers [16], [19], [21], [23], [26], [30], [39], [42] are a kind of hardware prefetchers and have been deployed by many processors due to low hardware complexity and substantial performance gains. They fundamentally exploit the regularity in the layout of data objects and the repetitiveness in access behaviors to predict future accesses. A vector called bit-pattern is used to record the access footprint within a fixed-size region (an address space consisting of several consecutive blocks). In addition, they also record the trigger event of the trigger access, i.e., the first access to the region. For example, "program counter (PC)" is a common trigger event. After recording, the prefetcher stores the pattern in a history table and sets the trigger event as the index. If an instruction with the same PC accesses a new region, the prefetcher will look up the corresponding access pattern in the history table using this PC. The prefetcher then preloads the data according to the access footprint indicated by this pattern.

Bit-pattern-based prefetchers fundamentally exploit the regularity in the layout of the data objects and the repetitiveness in their access behavior to predict future accesses. They use a vector that is called bit-pattern to record access footprint within a fixed-size region (an address space consisting of several consecutive blocks). In addition, they also record the trigger event of the trigger access, i.e., the first access to the region. For example, "PC" is a common trigger event. After recording, the prefetcher stores the pattern in a history table and sets the trigger event as the index. If an instruction with the same PC accesses a new region, the prefetcher will look up the corresponding access pattern in the history table using this PC. The prefetcher then preloads the data according to the access the footprint indicated by this pattern.

Manuscript received 2 August 2024; accepted 3 August 2024. This work was supported in part by the National Natural Science Foundation of China under Grant 92373205; and in part by the National Key Research and Development Program of China Grant 2023YFB4404400; and in part by the Alibaba Group through Alibaba Innovative Research Program. This article was presented at the International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding authors: Zonghui Wang; Wenzhi Chen.)*

Yiquan Lin, Wenhai Lin, Jiexiong Xu, Zhen Jin, Jingchang Qin, Jiahao He, Zonghui Wang, and Wenzhi Chen are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: zhwang@zju.edu.cn; chenwz@zju.edu.cn).

Yiquan Chen, Shishun Cai, and Yuzhong Zhang are with the Cloud Infrastructure Service, Alibaba Group, Hangzhou 310030, China.

Digital Object Identifier 10.1109/TCAD.2024.3442981

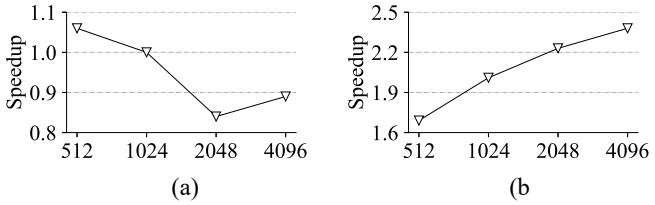

Fig. 1. Performance of two different phases in 429.mcf with various region sizes (a) 429.mcf-184B (b) 429.mcf-217B.

The region size is one of the most crucial parameters of bit-pattern-based prefetchers, which denotes the size of the memory that can be recorded by a pattern and prefetched by a prediction. The region size reflects the working granularity of bit-pattern-based prefetchers and impacts both the performance and hardware overhead.

All the existing bit-pattern-based prefetchers (e.g., SMS [39], Bingo [16], DSPatch [19], and PMP [26]) only support one fixed region size, typically 2 or 4 KiB. However, this rigid approach results in suboptimal performance and wastes hardware resources for various applications. To quantify the effects of the region size, we evaluated the performance of numerous applications with different region sizes in the simulator. Fig. 2 demonstrates that different applications have unequal optimal region sizes. The optimal region sizes are also not the same in different phases for one application due to the variations of data structure sizes and memory access patterns, as shown in Fig. 1(a) and (b). With fixed the region size, applications may face performance bottlenecks because the prefetcher cannot select the optimal region size for various applications. Meanwhile, the fixed region size results in huge hardware resource consumption. We evaluated the impact of region size on the hardware overhead in Section III-B. When the region size is too small, some patterns may need to be stored in two entries, resulting in additional hardware overheads, which can be alleviated by adopting a larger region size. On the other hand, when the region size is too large, some patterns exhibit all the zeros in the half of the regions, causing high hardware overheads. Prefetchers can avoid storing this long string of zeros by adopting a smaller region size.

In this article, we propose PARS, a pattern-aware spatial data prefetcher supporting multiple region sizes. PARS supports prefetching data with multiple region sizes, achieving application performance enhancement, and hardware overhead reduction. In addition, we propose an adaptive RS-switching mechanism to dynamically adopt appropriate region sizes for various applications and phases of the application. The PARS can automatically set the appropriate region size based on the historical accuracy information of the prefetcher and the region footprint. We implemented PARS on Champsim [9] simulator and evaluated it on 198 traces from SPEC CPU 2006 [12], SPEC CPU 2017 [13], Ligra [37], and Cloudsuite [22]. Results show that PARS outperforms the two state-of-the-art prefetchers Bingo [16] by 2.1% (up to 24.4%) and Pythia [18] by 3.9% (up to 111.2%) in the single-core system. In the four-core system, PARS outperforms Bingo by 5.0% (up to 66.0%) and Pythia by 5.4% (up to 177.9%) with only 17.5% overhead of Bingo.

We make the following contributions in this article.

1) We identify the impact of region size on the bit-pattern-based prefetchers and observe the drawbacks of the existing single region size architecture: the single region size architecture faces performance bottlenecks for various workloads and high hardware overhead.
2) We propose a novel pattern-aware spatial data prefetcher that supports multiple region sizes. Meanwhile, we propose an adaptive RS-switching mechanism to dynamically adopt appropriate region sizes for various applications and phases of the application.
3) We implement PARS and evaluate it on 198 traces from the four benchmark suites, PARS outperforms the four state-of-the-art prefetchers. In particular, PARS outperforms Bingo by 5.0% and Pythia by 5.4% with only 17.5% overhead of Bingo.

II. Background

A. Bit-Pattern-Based Prefetchers

The bit-pattern-based prefetcher is an essential type of spatial data prefetcher, which adopts the bit vector to record the footprints of a fixed-size memory (region) and prefetch data. Each bit in the vector records whether a cache block is accessed or not during the training period, where `1` means it is accessed and `0` means it is not accessed. For instance, a 64-bit vector can represent the footprint of a 4 KiB region consisting of 64 consecutive cache blocks (typically 64 bytes).

During training, each vector will record the footprint starting with the first access to the corresponding region and ending with the eviction of any block in the region from the cache. The prefetcher also records the trigger event which is extracted from trigger access, i.e., the first access to the region. There are five common trigger event types: PC, "Offset," "Address," "PC+Offset," and "PC+Address." For example, the trigger event of PC+Offset indicates the address of the instruction and the ordinal position of the accessed cache block within this region.

After recording, the prefetcher inserts the ⟨*event*, *pattern*⟩ pair into a pattern history table (PHT). When the same trigger event occurs, the prefetcher uses it to look up the corresponding pattern and preloads the cache blocks according to the access footprint indicated by this pattern. For example, the prefetcher adopts PC+Offset as the trigger event. An instruction `I` accessing `X+1` (the second cache block of region `X`) triggers the recording for the region `X`. Then, the applications only access the block `X+3` of this region. As a result, the pattern for the trigger event of instruction `I` and offset `1` is `0101`. After recording, if instruction `I` accesses `Y+1`, the prefetcher will find the pattern `0101` based on the instruction `I` and the offset `1`, and then prefetch the block of `Y+3`. Note that, the block `Y+1` of trigger access is not prefetched.

B. Common Framework of Bit-Pattern-Based Prefetchers

Modern bit-pattern-prefetcher [16], [26], [39] includes three primary components: 1) a filter table (FT); 2) an accumulation table (AT); and 3) a PHT. The FT records regions with only one cache block accessed, which can not trigger any data

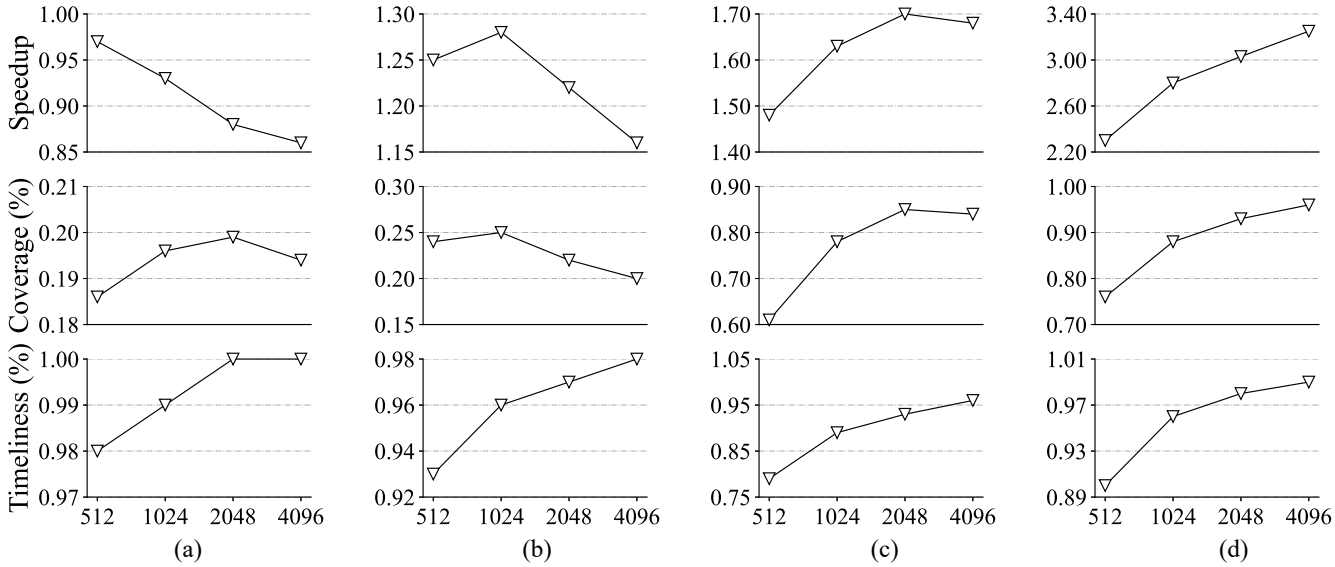

Fig. 2. Performance, coverage, and timeliness of four traces with various region sizes. (a) Ligra_PageRankDelta-24.5B. (b) 450.soplex-92B. (c) Ligra_BC-17B. (d) 630.bwaves_s-891B.

prefetching. Then, the second access to the region in the FT will activate its recording phase and transfer the entry from FT to AT. The AT will keep track footprints of all the regions until the eviction of any block in the region from the cache. So far, the prefetcher finishes the training of one region and inserts the pair ⟨*event*, *pattern*⟩ of the region into the PHT. All the three tables are essentially set-associative caches and are managed by a replacement algorithm (e.g., LRU).

C. Dual Trigger Events Design

To achieve high accuracy and high matching probability simultaneously, emerging prefetcher [16] supports dual trigger events on PHT, which can look up patterns based on PC+Address or PC+Offset. During training, whenever inserting a new pattern to the PHT, the prefetcher uses PC+Offset to find a cache set. Then, the pattern is inserted into the set and tagged by PC+Address. In prediction, the prefetcher first looks up with PC+Address because it has higher accuracy. If a matching pattern is found, it will be used to issue prefetches. Otherwise, the prefetcher looks up with PC+Offset. In this case, the prefetcher only matches the PC+Offset bits in the tag and the other bits are masked.

III. Motivation

Embedded systems, such as those in autonomous vehicles and industrial autonomous devices require extremely high performance to process the large volumes of the real-time data. Meanwhile, they are also constrained by limited hardware resources. However, existing prefetching techniques do not adequately address these demands and incur substantial storage overhead.

In this section, we analyse the impact of region size on the prefetching performance (Section III-A) and storage overhead (Section III-B), respectively. We found that the fixed region size of the existing schemes cannot meet the requirements for all various applications, resulting in performance loss and high storage overhead. This motivates us to design a novel prefetcher that supports multiple region sizes.

A. Impact of Region Size on Prefetching Performance

To evaluate the performance impact of the region size, we ran 198 traces on Bingo with various region sizes from 0.5 to 4 KiB. The storage overhead of configurations with different region sizes are all comparable. Specifically, with each halving of the region size, the number of entries for both AT and PHT is doubled.

Fig. 2 illustrates three metrics (i.e., speedup, coverage, and timeliness) for the four different applications in different region size configurations. First, the data on speedup demonstrates that different applications have different optimal region sizes. Specifically, trace “450.soplex-92B” achieves the maximum speedup when the region size is set to 1 KiB. While the trace “Ligra_BC-17B” achieves the maximum speedup as the region size is set to 2 KiB. Second, the coverage is also significantly affected by the region size, which is consistent with the trend of the speedup except for trace “Ligra_PageRankDelta-24.5B.” Finally, the timeliness improves with increasing region size, because prefetchers with large region sizes can predict larger memory space each time. As a result, various applications have diverse sensitivity to region size changes. At the same time, Fig. 1(a) and (b) show that the optimal region size varies in different phases for one application due to the difference in the data structure sizes and memory access patterns. Therefore, bit-pattern-based prefetchers with a fixed region size can not meet the requirements for numerous applications and lead to suboptimal performance.

There are significant performance variations in different region size configurations because every application has its optimal region size. Numerous factors influence the optimal region size, such as the size of data structures, program access behaviors, etc. When the region size falls below the optimal

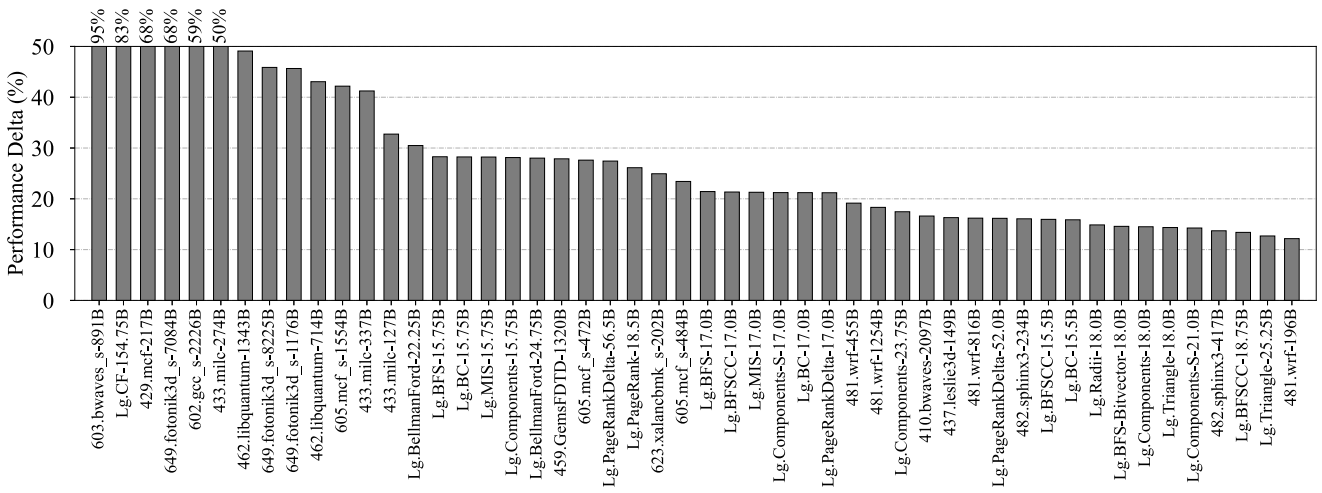

Fig. 3. Performance delta of the top 50 traces.

region size, the performance improves as the region size increases. For example, Fig. 2 (b), (c), and (d) have optimal region sizes of 1, 2, and 4 KiB, respectively. Larger region sizes in these cases exhibit enhanced timeliness and fewer trigger accesses that cannot be prefetched, resulting in better performance.

On the other hand, performance begins to decline when the region size surpasses the optimal region size in Fig. 2 (a), (b), and (c), for the two following reasons.

1) Large region sizes are more prone to encompass unrelated data structures. Consequently, many accesses are unduplicated with respect to the trigger access, resulting in pronounced overprediction. For instance, in the case of 450.soplex-92B, we found the increase in region size correlates with a substantial rise of 27% in useless prefetches, thereby consuming the bandwidth, contaminating the cache, and ultimately deteriorating performance.
2) When the storage overhead of configurations with different region sizes is comparable, prefetchers with larger region sizes have fewer entries, resulting in poor performance. When a program accesses only a small subregion within a larger region, prefetchers with different region size configurations all allocate one AT entry and one PHT entry. However, in our setups, prefetchers with larger region sizes have fewer entries compared to those with smaller region sizes. As a result, they exhibit inferior training and storage capabilities, leading to decreased performance.

To more intuitively show how much the region size affects performance, we define the performance delta for each trace as the difference between the maximum and minimum speedup in the four region sizes. Fig. 3 shows the performance delta of the top 50 traces. Trace "603.bwaves_s-891B" has the largest performance delta of 95%. Moreover, the average performance delta of the top 50 traces is 29.5%, indicating that most real-world applications are very sensitive to the region size, and improper region size would cause severe performance degradation.

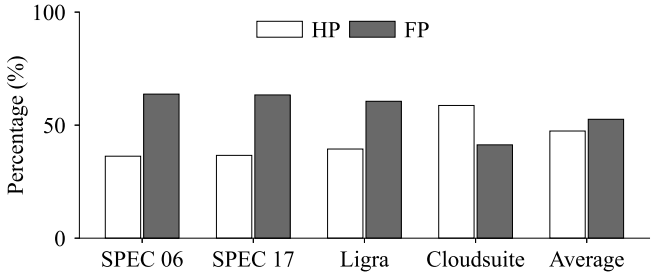

Fig. 4. Percentage of HP and FP. HP is the pattern where the first half or the second half is all 0 and FP is the other pattern.

B. Impact of Region Size on Storage Overhead

Inappropriate region size also causes hardware inefficiency and extra overhead. We recorded all the patterns captured by Bingo with 4 KiB region size in 198 traces and classified them into the half pattern (HP) and the full pattern (FP). In particular, HP is the pattern where the first half or the second half is all "0" and FP is the other pattern. We counted the number of HP and FP, respectively, and the results are shown in Fig. 4.

For 4 KiB region size, we found that Cloudsuite has the maximum HP percentage of 59% and the HP accounts for 43% of all patterns on average, which indicates that the prefetcher stores many "half 0s." The half 0s do not contribute to prefetching and waste one-third of a 4 KiB entry's storage. Therefore, the 4 KiB region size causes extra storage overhead due to HPs. To address this issue, adjusting the region size to 2 KiB can eliminate the half 0s' in 4 KiB region size. However, using 2 KiB region size would introduce another issue. Notice that, entries contain not only data fields such as patterns, but also metadata fields, such as tag and LRU. The 2 KiB prefetcher must allocate two entries to store each FP in Fig. 4. Because of double metadata for two entries, the storage overhead of two 2 KiB entries is also one-third more than a 4 KiB entry. Therefore, using only 2 KiB region size also results in additional storage overhead due to extra tags and LRUs.

As a result, the prefetchers with fixed region sizes face additional storage overhead. This motivates us to design

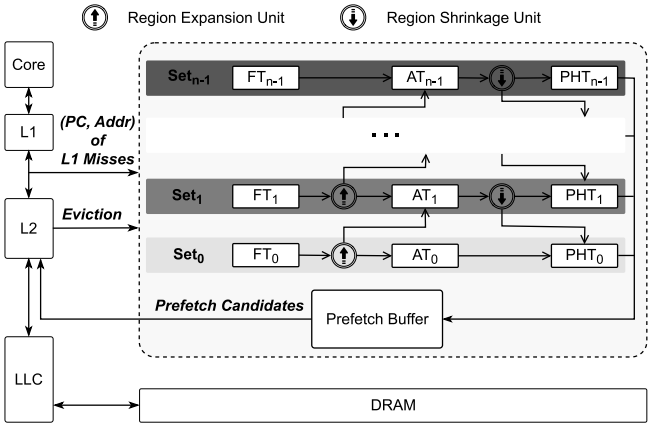

Fig. 5. Overview of system with PARS prefetcher.

PARS that supports multiple region sizes to minimize storage overhead. The idea behind PARS is to replace a portion of entries of a single region size with entries of other region sizes. For example, for a 2 KiB table with 4K entries, PARS tries to replace 2K of those entries with 1K 4 KiB entries and stores HPs and FPs separately. The replaced entries can reduce the storage overhead while ensuring the same storage capacity for FPs.

IV. Design

In this section, we describe the design of PARS, a pattern-aware spatial bit-pattern-based prefetcher that supports the multiple region sizes and can dynamically adopt appropriate region sizes for various applications through an adaptive RS-switching mechanism.

A. Overview

Architecture: Fig. 5 depicts the overall architecture of PARS. The key idea of PARS is that it supports multiple region sizes, enabling it to simultaneously enhance the application performance while reducing the hardware overhead. For each region size, PARS utilizes a dedicated set of AT, FT, and PHT for training and prediction purposes. These table sets are denoted as Set$_i$($0 \leq i < n$), where n is the number of region sizes PARS supports. The PHTs adopt dual trigger events design as described in Section II-C. Besides, there is a prefetch buffer (PB) that uses Set$_{n-1}$'s region size and stores the pattern of the region to be prefetched. To achieve dynamic region size switching, we introduce the region expansion unit (EU) and shrinkage unit (SU) for each table set. The EU is located between FT and AT, which detects whether there are mergeable regions. If the detection meets the requirements, it will merge two entries in Set$_i$ and send the new entry to AT$_{i+1}$. In contrast, the SU is located between AT and PHT, which detects the half pattern (see Section III-B) and the prediction accuracy. If the detection meets the requirements, it would shrink the region in Set$_i$ and send the pattern to PHT$_{i-1}$. In this way, PARS supports multiple region sizes and can dynamically adopt appropriate region sizes for applications.

Training: PARS starts training a region when the program accesses a new region. Whenever an L1 miss comes, PARS looks up FTs and ATs of all the table sets. If there is no corresponding entry in all FTs and ATs, which indicates that it is a trigger access, PARS will start training the region. Fig. 6 illustrates an example of the training process of access patterns of a new region, which consists of four steps.

1) When the program accesses a new region with a cache line address of `r+1`, PARS assigns a new FT entry for the region `r` with the offset `1` and the corresponding `pc`. By default, PARS will insert the new FT entry into a random Set, as PARS can switch it to an appropriate region size effectively.
2) When another access to the address `r+3` in the same region `r` with another offset of `3`, PARS delivers the FT entry to the AT$_{n-1}$. Then, the AT$_{n-1}$ allocates a new AT entry containing a pattern field that is initialized with the first two accesses to the region (`01010000`). If it were not in Set$_{n-1}$, the FT entry would be sent to the EU first rather than the AT, which checks whether this entry needs to be expanded or not (detailed in Section IV-B).
3) Next, every subsequent access (e.g., `r+5`) to the region will update the pattern (`01010100`) in the AT entry.
4) The AT will stop tracking the footprint of the region `r` when either of the following two cases happens. (a) Any block in the region `r` is evicted from the cache and (b) the AT entry of the region `r` is evicted due to the allocation of a new AT entry.

After stopping tracking the footprint, the AT entry is sent to the SU, which checks whether this entry needs to be shrunk or not (detailed in Section IV-B). Then, the pattern of the region `r` is stored in the PHT$_{n-1}$, which is indexed by PC+Offset (`pc+1`) and tagged by PC+Address (`pc+r+1`). So far, the training for the region `r` is finished.

Prediction: PARS predicts the footprints for a region when the program accesses a new region. Fig. 7 illustrates an

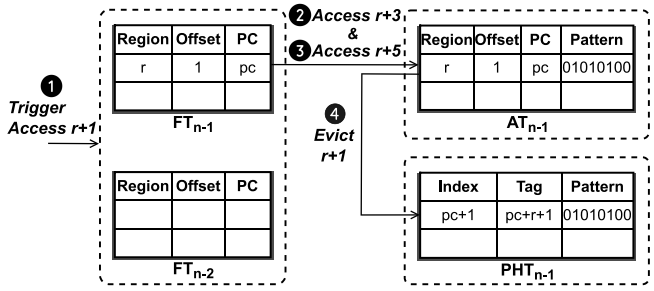

Fig. 6. Training process of a region.

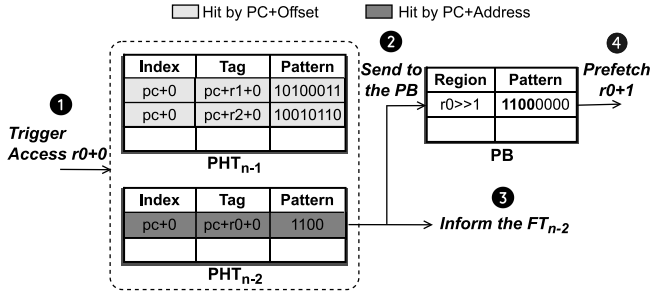

Fig. 7. Prediction process of a region.

example of the prediction process, which consists of three steps.

1) When an instruction pc accesses a new region with a cache line address of r0\ensuremath{+}0, PARS looks up PHTs of all table sets in parallel. Each PHT has three possible results: "Hit by PC+Address," "Hit by PC+Offset," and "Miss." In the example in Fig. 7, the query with index pc+0 hits both the PHT_{n-1} and the PHT_{n-2}. The result of PHT_{n-2} is Hit by PC+Address while the result of PHT_{n-1} is Hit by PC+Offset. PARS prefers the result of Hit by PC+Address to Hit by PC+Offset because of its higher accuracy. If the results of multiple PHTs are Hit by PC+Offset, PARS gives higher priority to the result from the PHT with a larger region size. Otherwise, PARS doesn't generate a prediction for the region if the results of all PHTs are Miss.
2) The predicted pattern will be sent to the PB and indexed by the currently accessed region. Since, PARS only has one PB that uses Set_{n-1}'s region size, the pattern and the region number from the PHT_{n-2} should be aligned to the PB's region size. The pattern 1100 should be expanded to 11000000 and the region number r0 should be shifted to match the PB's region size.
3) Inform the FT in the table set where the result is adopted to initiate the training of the region instead of the default table set. In addition to sending the pattern to the PB, a hit on the PHT lookup process also impacts the training. The table set, where the pattern is adopted, instead of a random table set is responsible for initiating the training of the new region, i.e., the FT_{n-2} would assign a new entry for the address r0+0.
4) PARS looks up the PB and prefetches the corresponding blocks. For every program access and not just for trigger access, PARS would look up the PB based on the currently accessed region and prefetch relevant blocks. With the PB, PARS can easily constrain the prefetching aggressiveness.

B. Adaptive RS-Switching Mechanism

We design the EU and SU in PARS and propose the adaptive RS-switching mechanism to enable the dynamical region sizes switching for different patterns.

Expansion Unit: We design the EU between FT and AT, which converts and transfers entries to a table set with a larger region size. When two mergeable regions are trained in Set_i at the same time, the EU switches their region sizes to Set_{i+1}'s. PARS considers two regions to be mergeable when 1) the two regions are adjacent and 2) the new region merged by the two regions is aligned.

Fig. 8 demonstrates the process of one region expansion, which consists of the following steps. Taking the access to the region 0x80 with the offset 2 as an example, we assume that the region 0x80 with the offset 1 is in the FT and its mergeable region 0x81 is in the AT.

1) After the access to region 0x80 with the offset 2, the PARS tries to send the FT entry to the EU.

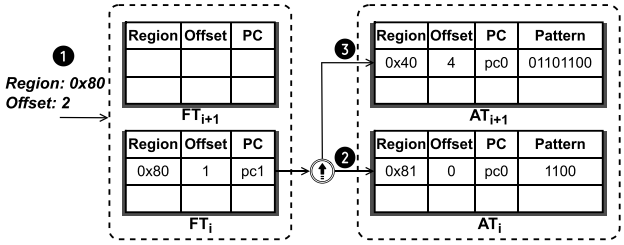

Fig. 8. Work process of the region EU.

2) The EU detects whether there are mergeable regions in the AT_i. It looks up the entry of region 0x81 in the AT_i. If the entry of region 0x81 is found, the EU merges the two entries into one and sends it to the AT_{i+1}. Otherwise, the entry of region 0x80 would be sent to the AT_i, and no further steps.
3) The EU merges the two mergeable entries into a larger one and sends the new entry to the AT_{i+1}.

If the EU decides to merge two entries, the four fields (region, offset, PC, and pattern) of the new AT_{i+1} entry have to be calculated based on these two entries.

1) The new region number can be obtained by dividing the original region number by two since the region size is doubled. In our example, the region field is set to 0x40, i.e., 0x80 divided by 2.
2) The new offset field is set based on the trigger access of the entry in the AT, which is earlier accessed. The new offset is the offset of the trigger access in the new region, which is set to 4 in the example. We take the Offset 0 in the region 0x81 entry as the trigger access. Meanwhile, the offset of the trigger access is 4 in the new region.
3) Similarly, the new PC field is set the same as the PC field of the entry in the AT. We set the PC as pc0 based on the trigger access of region 0x81.
4) The new pattern field is set to 01101100 by splicing the two patterns of region 0x80 and region 0x81 in order. In this way, the EU effectively merges region 0x80 and region 0x81 and switches the region size to a larger one.

Shrinkage Unit: We design the SU between AT and PHT to shrink regions to a smaller size, which eliminates half 0s and enhances the prefetching accuracy. When the $AT_i (0 < i < n)$ finishes recording an entry, it will send the entry to the SU. The SU determines whether to split and insert new entries to the table set with a smaller region size.

PARS would shrink the region when 1) the entry contains a half 0 or 2) the prediction accuracy of the half region that does not contain the trigger access is lower than the threshold. The prediction accuracy can be obtained by calculating the difference between the actual access footprint and the predicted footprint. Specifically, the actual access footprint is the pattern coming from the AT, and the predicted footprint is the pattern stored in the PHT with the same PC+Address (if it exists).

Fig. 9 illustrates an example of region shrinkage due to the presence of half 0.

1) When the AT_i finishes recording region 0x22, it will send the entry to the SU. Since the pattern 00001010

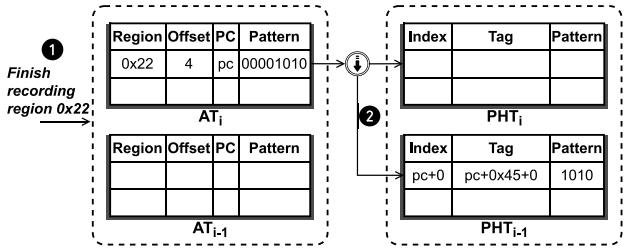

Fig. 9. Work process of the region SU. The region is shrunk due to the presence of half 0.

has a half 0, the SU decides to switch its region size to a smaller one.

2) When the SU decides to shrink the region from the AT$_i$, it updates three fields (region, offset, and pattern) and stores the new pattern into the PHT$_{i-1}$.
3) The new region number field is set to twice the origin region number and its lowest bit is set to `0` or `1` based on the reserved half pattern. In our example, the lowest bit is set to `1`, so the region number is updated from `0x22` to `0x45`.
4) The new offset field is set to the offset of the trigger access block in the new small region. The offset in the example is set to `0`.
5) The new pattern is set to `1010`, which is the half of the original pattern containing the trigger access. Finally, the new pattern is stored in the PHT$_{i-1}$, which is indexed by PC+Offset (`pc+0`) and tagged by PC+Address (`pc+0x45+0`). As a result, the SU effectively shrinks the original region into an appropriate one.

C. Region Size Selection

The architecture diagram of PARS depicted in Fig. 5 accommodates `n` region sizes, necessitating the precise selection of the appropriate region sizes. Since, the RS-Switching mechanism achieves dynamic switching between the neighboring region sizes by the EU and SU, these `n` sizes need to follow a progression of consecutive powers of two. In addition, in systems with a 64B block size and 4 KiB page size, the region size should fall within the range of [0.125, 4] (KiB). We evaluated the performance of the whole region size configurations as shown in Fig. 20. For the best tradeoff between the performance enhancement and hardware complexity, the implementation of PARS in this article adopts the [2, 4] (KiB) configuration as detailed in Section V-G.

V. Evaluation

A. Experimental Setup

We used Champsim [9] to evaluate PARS. Champsim is a trace-driven simulator that has been used for the second and third data prefetching championships (DPC-2 [2] and DPC-3 [3]). We list the simulation parameters in Table I. In both single-core and multicore evaluations, we used the first 50 M instructions to warmup and the next 200 M instructions to simulate. We report the performance in terms of the IPC improvement (speedup) over a baseline without any prefetcher.

TABLE I
Simulator Parameters

Core	Out-of-order, 4-wide fetch, 256-entry ROB 72-entry LQ, 56-entry SQ
L1I	private, 32 KiB, 8-way, 4-cycle, 16-entry MSHRs
L1D	private, 32 KiB, 8-way, 4-cycle, 8-entry MSHRs
L2C	private, 256 KiB, 8-way, 10-cycle, 32-entry MSHRs 32-entry RQ, 32-entry WQ, 32-entry PQ
LLC	1-core: 2 MiB, 4-core: 8 MiB, 16-way, 20-cycle 64 MSHRs per LLC Bank, 32-entry PQ
DRAM	1-core: 1 channel, 4-core: 2 channels 2400 MTPS, 8B channel width tRP=15ns, tRCD=15ns, tCAS=12.5ns

TABLE II
Configurations of Five Prefetchers

Prefetchers	Configurations	Overhead
Pythia	The same as [18]	25.5 KB
MLOP	128-entry AMT, 16-prefetch_degree	8.5 KB
Bingo	2 KiB region, 64-entry FT, 128-entry AT 16K-entry PHT, 128-entry PB	121.8 KB
PMP	4 KiB region, the same as [26]	4.3 KB
PARS	[2, 4] (KiB) regions, the same as Table IV	21.3 KB

Workloads: We used 198 traces from the four benchmark suites, including SPEC CPU 2006 [12], SPEC CPU 2017 [13], Ligra [37], and Cloudsuite [22]. For SPEC CPU 2006 and SPEC CPU 2017, we reused the traces provided by DPC-2 and DPC-3. For Ligra, we used the traces provided by Pythia [18]. For Cloudsuite, we reused the traces provided by CRC-2 [1]. In all the evaluations, we ignored traces whose LLC miss per kilo instructions (MPKI) is less than 1 because all the prefetchers have similar performance improvements for these traces.

Prefetchers: We compared PARS with four prior prefetching proposals: 1) Pythia [18]; 2) MLOP [35]; 3) Bingo [16]; and 4) PMP [26]. Pythia is an emerging prefetcher that employs reinforcement learning. MLOP is an excellent offset prefetcher, which is one of the winners in DPC-3. Bingo is one of the state-of-the-art spatial prefetchers which is based on SMS [39]. PMP is the latest lightweight bit-pattern-based prefetcher that employs the strategy of merging similar patterns. To be fair, we placed all the prefetchers on L2 cache (L2C) and no other prefetchers in L1 cache (L1C) or LLC. All the prefetchers were trained on L1C miss and fill the prefetched cache lines into L2C and LLC. Table II shows the configurations of the five prefetchers.

B. Single-Core Performance

Fig. 10 shows the performance of five prefetchers in a single-core system, indicating that PARS surpasses the performance of the other four prefetchers. On average, PARS improves performance by 40.6% (up to 342.5%) over the baseline without a prefetcher and outperforms Pythia, MLOP, Bingo, and PMP by 3.9% (up to 111.2%), 5.9% (up to 87.5%), 2.1% (up to 24.4%), and 3.3% (up to 57.4%), respectively. For Cloudsuite, the performance of all the prefetchers is similar, due to the majority of workloads exhibiting low MPKI.

PARS outperforms Pythia and MLOP by leveraging deep prediction. Pythia and MLOP struggle to prefetch deeply since they use "delta" features to make predictions. In contrast,

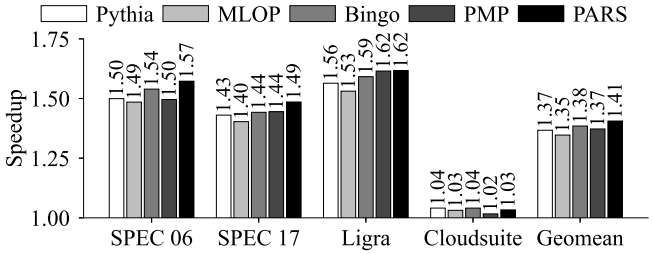

Fig. 10. Single-core performance of five prefetchers.

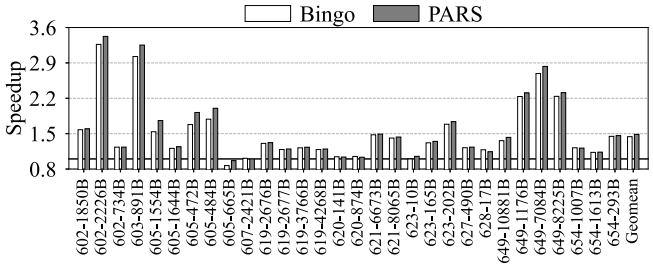

Fig. 11. Single-core performance of Bingo (state-of-the-art prefetcher) and PARS on SPEC CPU 2017.

PARS can generate a few dozen blocks in one prediction. Benefiting from the deep prefetching, PARS has better timeliness and outperforms Pythia and MLOP by more than 6% for all the benchmark suites except Cloudsuite. PMP employs an unstable strategy, including merging and extracting patterns. When its extracting precision is poor, such as in SPEC CPU 2006, its performance is much lower than PARS by 7.7%.

Compared to Bingo, PARS can adaptively adjust the region size according to the memory pattern of workloads. Thus, PARS performs better than Bingo on most benchmarks and consumes only 17.5% storage of Bingo. Since, Bingo is a state-of-the-art prefetcher, we have a head-to-head comparison of Bingo and PARS for each trace in SPEC CPU 2017 as shown in Fig. 11. For SPEC CPU 2017, PARS outperforms Bingo by 5.3% on average. On the majority (25 out of 30) of traces, PARS has better performance improvement than Bingo. Specifically, PARS outperforms Bingo by 24.0% on "605.mcf_s-472B."

C. Prefetching Metrics

Coverage and Overprediction: Are both important metrics for prefetching performance. Coverage is the ratio of reduced load misses relative to total load misses of the baseline with no prefetcher while the overprediction is the ratio of increased read misses relative to total read misses of the baseline with no prefetcher.

Fig. 12 shows the metrics of each prefetcher across all the benchmark suites in the single-core system. On average, PARS offers 8.5%, 15.9%, 5.5%, and 26.8% higher coverage than Pythia, MLOP, Bingo, and PMP, respectively. The highest coverage is an important cornerstone for PARS to gain the optimal performance improvement. Meanwhile, the overprediction of PARS is 1.7%, 19.5%, and 256.4% lower than MLOP, Bingo, and PMP, respectively.

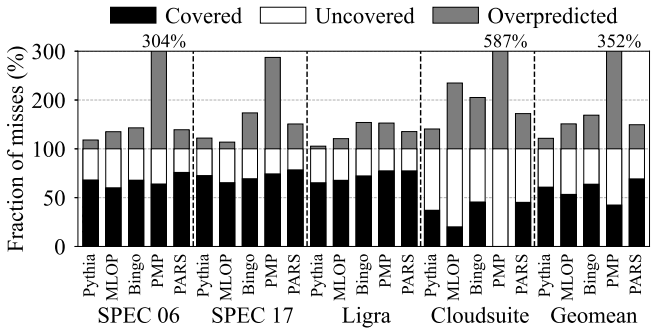

Fig. 12. Coverage and overprediction of five prefetchers.

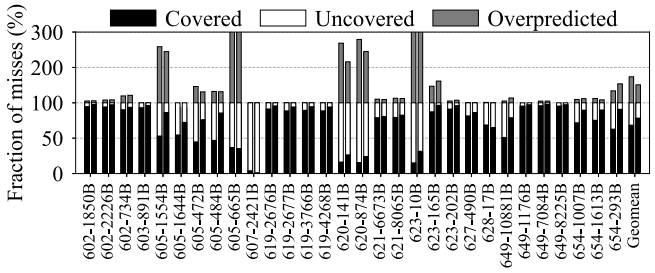

Fig. 13. Coverage and overprediction of Bingo and PARS on SPEC CPU 2017. For each trace, the left bar is Bingo and the right bar is PARS.

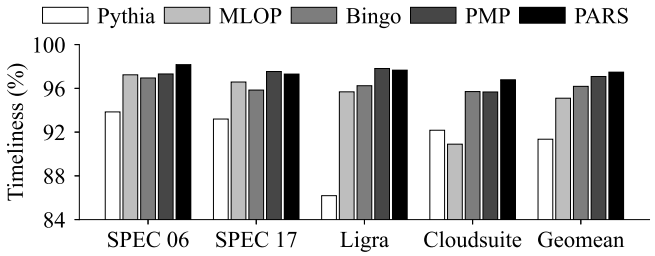

Fig. 14. Timeliness of five prefetchers.

Fig. 13 shows a head-to-head comparison between Bingo and PARS for each trace on SPEC CPU 2017. On the majority (27 out of 30) of traces, PARS exhibited enhanced coverage (up to 39%). On average, PARS boosted coverage by 10%, and decreased overprediction by 23%.

Timeliness: A useful prefetch should ensure that the data is filled into the cache before it is accessed; otherwise, it is considered a late prefetch. We define timeliness as the ratio of useful prefetches to the total of useful prefetches and late prefetches. The results of five prefetchers are shown in Fig. 14. We observe that both PARS and PMP have excellent timely rates that are all greater than 97% because they can learn patterns for 4 KiB regions and issue up to 63 prefetches at a time. Pythia has the worst timeliness. In Ligra, the timeliness of Pythia is only 86%.

DRAM Traffic: We define the additional DRAM traffic (ADT) as the ratio of increased DRAM accesses to those in the baseline. Fig. 15 shows the ADT of the five prefetchers. We can observe that the ADT of PARS is lower than that of MLOP, Bingo, and PMP, indicating that PARS consumes less memory bandwidth and achieves better performance. PARS is more aggressive than Pythia and has a little higher ADT. Increasing the prefetch degree of Pythia can make it as aggressive as

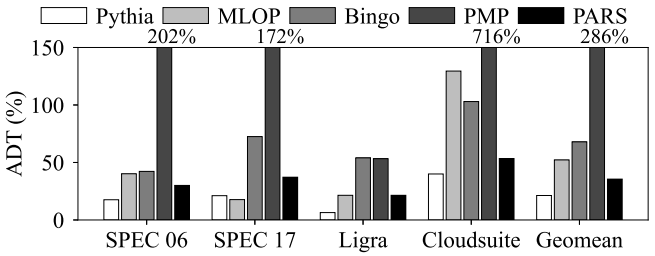

Fig. 15. ADT of five prefetchers.

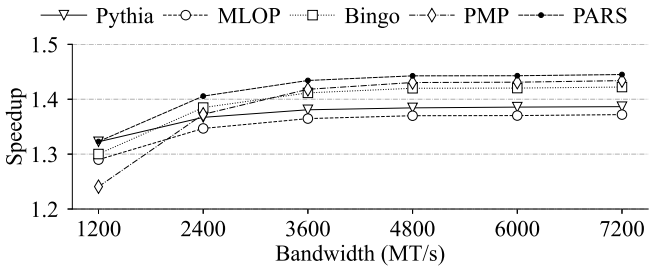

Fig. 16. Performance scaling with DRAM bandwidth.

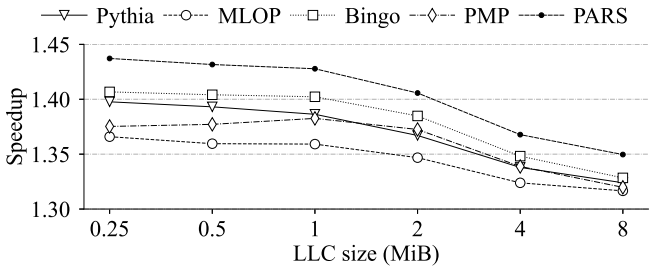

Fig. 17. Performance scaling with LLC size.

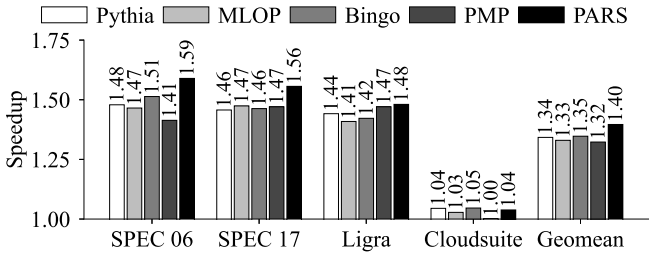

Fig. 18. Multicore performance of five prefetchers.

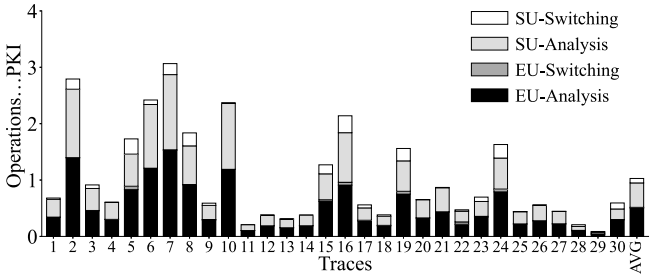

Fig. 19. Frequency of analysis and switching for EU and SU, measured in occurrences PKI.

PARS. PARS still outperforms the aggressive Pythia by 1.2% with 6.9% lower ADT. On the other hand, simply controlling the prefetching degree of PB can limit the aggressiveness of PARS. The limited PARS reduces 10.3% ADT with 1.4% performance loss, which still outperforms Pythia by 2.5%. The simple mechanism of limiting prefetching aggressiveness through PB can effectively reduce ADT, making PARS more suitable for the embedded systems.

D. Sensitivity in Single-Core System

Varying DRAM Bandwidths: Fig. 16 shows how the performance improvements of all the prefetchers change when we scaled the DRAM bandwidth from 1200 to 7200 MT/s. We observe that PARS gains the highest performance improvement in all the bandwidth configurations. In a low-bandwidth scenario at 1200 MT/s, PARS shows excellent adaptive ability and outperforms Pythia, MLOP, Bingo, and PMP by 0.1%, 3.3%, 2.3%, and 8.2%, respectively. PARS only has a slight advantage over Pythia because bit-pattern-based prefetchers have greater bandwidth requirements. As the bandwidth grows, PARS shows better performance and quickly pulls away from Pythia. When the bandwidth reaches 6000 MT/s, the performance of all the prefetchers stabilizes, and PARS outperforms Pythia, MLOP, Bingo, and PMP by 5.7%, 7.3%, 2.3%, and 1.1%, respectively.

Varying LLC Size: Fig. 17 shows the average speedup across the four benchmark suites when the LLC size varies from 0.25 to 8 MiB. We observe that PARS outperforms other prefetchers in all the LLC size configurations. When the LLC size is small, PARS exhibits greater advantages over the other prefetchers. Specifically, in 1 MiB configuration, PARS outperforms Pythia, MLOP, Bingo, and PMP by 4.1%, 6.9%, 2.6%, and 4.5%, respectively, indicating that PARS is better adapted to the environment where the LLC resources are highly competitive. When the LLC size is greater than 1 MiB, the performance of all the prefetchers declines as the baseline IPC increases rapidly. Nevertheless, PARS continues to outperform the other prefetchers. In 4 MiB configurations, PARS outperforms Pythia, MLOP, Bingo, and PMP by 3.0%, 4.4%, 2.0%, and 2.9%, respectively.

E. Multicore Performance

Fig. 18 shows the performance of five prefetchers in a four-core system. PARS outperforms Pythia, MLOP, Bingo, and PMP by 5.4%, 6.6%, 5.0%, and 7.4%, respectively. The advantages of PARS are more pronounced in multicore systems than in single-core systems. The main reason is that multiple workloads will compete for the DRAM bandwidth and LLC resources, and PARS adapts well to both the low-bandwidth and low LLC size scenarios. In SPEC CPU 2006 and 2017, PARS significantly outperforms the remaining four prefetchers by more than 8%. This is because the benchmarks in SPEC are diverse, and PARS's multiple region size architecture has a wider adaptability to various types of applications.

F. Region Analysis and Resizing Frequency

Fig. 19 shows how frequently the region is analysed and resized by the EU and SU of thirty traces. Across the four benchmark suites, the EU and SU analyse 1.82 and 1.80 times per kilo instructions (PKI), respectively, to determine

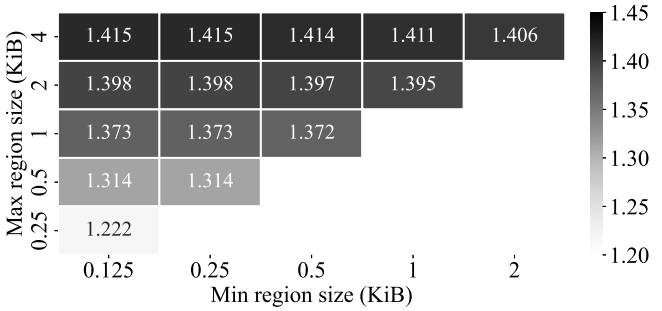

Fig. 20. Speedup with different region size configurations. The x-axis represents the minimum region size in the configuration, while the y-axis represents the maximum region size. For instance, the block (x, y) = (0.25, 2) shows the performance of configuration [0.25, 0.5, 1, 2] (KiB), which is 1.398.

TABLE III
OVERHEAD AND PERFORMANCE OF PARS WITH DIFFERENT PHT SIZES

PHT size	0.25K	0.5K	1K	2K	4K
Overhead (KiB)	6.6	11.6	21.3	40.5	78.5
Speedup	1.385	1.394	1.406	1.412	1.417

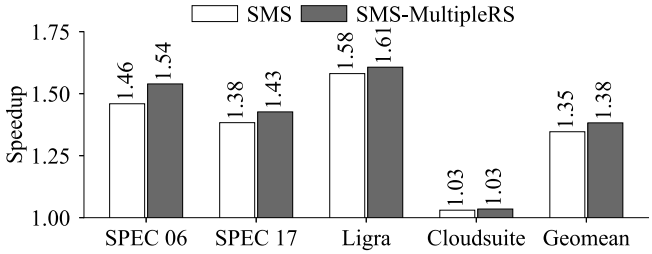

Fig. 21. Performance of original SMS and SMS with multiple region sizes.

whether to adjust the region size. On average, they perform further resizing operations at frequencies of 0.02 and 0.35 PKI, respectively. These frequencies are very low compared to the L2C MPKI in baseline, which is 9.00. For each analysis, EU and SU require one lookup for AT and PHT, respectively. The SU also needs simple operations, such as XOR and PopCount. For each resizing, EU and SU only need simple bitwise operations, which can be done in one cycle. The latency of EU and SU in the worst path is 6 and 14 cycles, respectively.

We employed a prefetcher without region size switching to evaluate the minimum time interval from when a pattern is trained to its first use across the four benchmark suites. On average, the minimum interval is 112 cycles, significantly exceeding the longest delay caused by region size switching. Additionally, the EU and SU are not on the critical path of prefetching (e.g., lookup the PB and issuing prefetch requests). Therefore, they do not decrease prefetching performance.

G. Preset Parameters

Region Sizes: PARS supports region sizes ranging from the two blocks up to the size of a page, and these sizes must be consecutive powers of two. To identify the optimal region size set, we evaluated the whole combinations of region sizes, totaling 15 combinations. The allocations of entries are comparable.

Fig. 20 illustrates the speedup for all the combinations. We make three key observations as follows.

1) Enhanced performance is achieved with larger maximum region sizes.
2) Performance improves with a greater variety of supported region sizes.
3) Adding smaller region sizes, particularly 0.125, 0.25, and 0.5 KiB, results in a marginal performance improvement of less than 0.1%.

When the maximum region size is set to 4 KiB (as shown in the top row of Fig. 20), PARS achieves the best speedup, at least 1.406. For the minimum region size choices, ranging from 2 to 0.125 KiB, PARS yields only a slight performance improvement. However, the increase in the number of levels leads to greater hardware complexity. For the best tradeoff between performance enhancement and hardware complexity, the implementation of PARS in this article adopts the [2, 4] (KiB) configuration. Additionally, for the integration into various processors, we can determine the optimal configuration of the PARS architecture according to the workloads.

PHT Sizes: PHT sizes represent the number of patterns that the PHT can store. We varied the PHT size to evaluate its impact on performance and overhead. We set the size of 2 and 4 KiB PHTs to be the same. Table III shows the overall overhead and performance of the prefetcher for each PHT size because PHT contributes the majority of PARS's storage, the prefetcher's overhead nearly doubles when the PHT size doubles. It is clear that the performance improves as the PHT size increases. Specifically, when the PHT size is increased from 512 to 1K, the performance improves most significantly by 1.1%. For the best tradeoff between the performance and overhead, we set PHTs' size to 1K.

H. Applying on Other Prefetchers

To further demonstrate the advantages of the multiple region sizes architecture, we applied the PARS design concepts to SMS [39] which is one of the most typical bit-pattern-based prefetchers. We named the new prefetcher SMS-MultipleRS. Both prefetchers use PC+Offset as the trigger event. SMS-MultipleRS has two table sets that can support both 2 and 4 KiB region sizes. Each PHT in SMS-MultipleRS has 1K entries and the PHT of SMS has 8K entries. Fig. 21 shows the performance of the two prefetchers. SMS-MultipleRS outperforms SMS by 3.6% on average, while the overhead is only 37.2% of SMS. We can conclude that multiple region sizes can effectively enhance the bit-pattern-based prefetchers regardless of their trigger events.

VI. DISCUSSION

A. Overhead Analysis

Table IV lists the details and storage overhead of each structure in PARS with the configuration of [2, 4] (KiB). In the default configuration of PARS, each FT, AT and PHT has 32, 32, and 1K entries, respectively, and the PB can store 16 patterns for 4 KiB regions. All tables are 16-way set associative and adopt LRU as replacement policies. The EU and SU do not require additional SRAM for data storage,

TABLE IV
DETAILS OF PARS'S STORAGE OVERHEAD

Structure	Width (bits)	Size	Storage
FT (2 & 4 KiB)	62 & 62	32 & 32	496 B
AT (2 & 4 KiB)	94 & 126	32 & 32	880 B
PHT (2 & 4 KiB)	63 & 95	1K & 1K	20224 B
PB	105	16	210 B
Total: 21.30 KiB			

TABLE V
AREA AND POWER OVERHEAD OF PARS

PARS compared to real systems	Area	Power
6-core, Ryzen5 4500, 65W TDP [5]	0.35%	0.13%
6-core, Ryzen Embedded v2546, 54W TDP [7]	0.35%	0.16%
64-core, EPYC 7H12, 280W TDP [4]	0.97%	0.32%

PARS's area: 0.09 mm^2/core; PARS's power: 14.01 mW/core
Bingo's area: 0.69 mm^2/core; Bingo's power: 83.89 mW/core

because they simply read data from FT, AT, or PHT, perform basic bitwise operations, and then insert the data into the corresponding tables. The total storage overhead of PARS is about 21.3 KB, which is only 17.5% of Bingo.

To accurately estimate PARS's hardware complexity, chip area, and power overheads, we used the Chisel [10] hardware design language (HDL) to implement the full-blown PARS, including all the tables, the EU and SU, and the control logic. For comparison, we also implement Bingo. We used Synopsys Design Compiler [14] and 7-nm library to estimate PARS's area and power overhead as shown in Table V. PARS consumes 0.09 mm^2 of area and 14.01 mW of power, which are, respectively, 13.04% and 16.70% of Bingo's values (0.69 mm^2 and 83.89 mW). Specifically, the EU and SU consume 691 combinational cells, whose area and power consumption account for just 0.08% and 0.38% of PARS, respectively, indicating the EU and SU have low hardware complexity.

Regarding the total die area and power consumption, the PARS implementation incurs minimal overheads as shown in Table V. Specifically, for a six-core Ryzen Embedded v2546 processor with 54 W TDP [7], PARS costs 0.35% and 0.16% of the area and power, respectively. We conclude that PARS improves performance with low area and power overhead.

B. Integrating Into Embedded Systems

Embedded system processors are typically constrained by strict requirements on latency, power consumption, and area. PARS can reduce cache misses to enhance the system performance and response speed, incurring only minor power and area consumption increases.

Since, embedded systems typically perform only specific tasks, it is critical to customize the optimal region size and entries for a particular embedded system. Determining the region size configuration based on the overhead analysis and performance evaluation within particular benchmarks is advisable. Furthermore, the number of entries needed depends on the code and data volume of the applications. For smaller-scale applications, reducing the size of each table can effectively decrease the hardware overhead. Finally, we can employ two dedicated registers to more flexibly manage PARS's impact on the system performance and power consumption. One register controls the enabling and disabling of PARS, and the other adjusts the prefetching aggressiveness (i.e., the prefetch degree of PB).

VII. RELATED WORK

To our knowledge, PARS is the first bit-pattern-based prefetcher that supports multiple region sizes. PARS can adaptively adjust the region size based on the current pattern and the past prediction precision. In Section V, we have compared PARS with some recent state-of-the-art prefetching techniques quantitatively. In this section, we compare PARS with the other relevant prefetching techniques.

Temporal Prefetchers: Temporal prefetchers [15], [17], [25], [27], [38], [43], [44], [46], [47] record the full block addresses of memory accesses. When a cache miss occurs, a temporal prefetcher will try to replay the historical miss sequence and issue prefetches for the subsequent addresses followed by the current address. Temporal prefetchers originated with the Markov prefetcher [27], which uses the fixed-size entries to store the address sequences. STMS [43] exploits variable-length temporal streams by utilizing a circular FIFO buffer. ISB [25] creates a structural address space and maps the physical addresses in a temporal stream into a continuous sequence of addresses, which can be prefetched by a simple next-line prefetcher. These temporal prefetchers are constrained by the large amount of metadata, which is usually multimegabytes and stored in the off-chip memory (DRAM). In contrast, PARS only requires 21.3 KiB and does not need to use the off-chip storage.

Spatial Prefetchers: The spatial prefetchers [16], [19], [20], [23], [24], [26], [28], [29], [31], [32], [33], [34], [35], [36] learn spatial correlations of access addresses rather than store full block addresses and have lower storage overhead than the temporal prefetchers. Emerging spatial prefetchers mainly learn the following two features as follows.

1) *Delta [28], [31], [32], [34], [35], [47]:* VLDP [36] can effectively enhance the performance of applications with multidelta sequences. SPP [28] creates signatures for address sequences and uses the signatures to predict the next delta. Sandbox [34] is an offset prefetcher that trains only one global delta from a predefined set. Moreover, BOP [31] builds on Sandbox by considering the timeliness of prefetching and learning a better delta. However, these prefetchers only issue one prefetch per prediction and have to use the strategy for deep prefetching recursively. In contrast, PARS learns the bit-pattern feature and can easily achieve deep prefetching.
2) *Bit-pattern [16], [19], [21], [23], [26], [30], [33], [39], [42]:* Ferdman et al. [23] adopted rotated bit-patterns to reduce the storage overhead. BuMP [42] enables bulk transfers by identifying high-density pages, which reduces energy consumption and improves throughput. DSPatch [19] learns two bit-patterns simultaneously by using AND and OR operations and selects them dynamically based on the bandwidth usage. Nevertheless, all these prefetchers can only learn bit-patterns with a fixed

region size. PARS supports multiple region sizes and can adaptively adjust the region size for each bit-pattern.

VIII. Conclusion

This article proposes PARS, a pattern-aware spatial data prefetcher supporting multiple region sizes. PARS supports multiple region sizes and dynamically switching appropriate region sizes for different patterns through an adaptive RS-switching mechanism. Evaluation results show that PARS can simultaneously enhance application performance while reducing hardware overhead and outperforms the state-of-the-art bit-pattern-based prefetcher.

References

[1] "2nd cache replacement championship." 2017. [Online]. Available: https://crc2.ece.tamu.edu

[2] "2nd data prefetching championship." 2015. [Online]. Available: http://comparch-conf.gatech.edu/dpc2

[3] "3rd data prefetching championship." 2019. [Online]. Available: https://dpc3.compas.cs.stonybrook.edu

[4] "AMD Epyc 7h12." 2020. [Online]. Available: https://www.x86-guide.net/en/cpu/AMD-EPYC-7H12-cpu-no7748.html

[5] "AMD Ryzen 5 4500." 2022. [Online]. Available: https://www.x86-guide.net/en/cpu/AMD-Ryzen-5-4500-cpu-no8468.html

[6] "AMD Ryzen embedded processors." Accessed: Jun. 14, 2024. [Online]. Available: https://www.amd.com/en/products/embedded/ryzen.html

[7] "AMD Ryzen embedded v2546." 2022. [Online]. Available: https://www.x86-guide.net/en/cpu/AMD-Ryzen-Embedded-V2546-cpu-no8426.html

[8] "Arm cortex m4 embedded processors." Accessed: Jun. 14, 2024. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m4

[9] "Champsim." 2023. [Online]. Available: https://github.com/ChampSim/ChampSim

[10] "The constructing hardware in a scala embedded language (chisel)." Accessed: Jun. 14, 2024. [Online]. Available: https://www.chisel-lang.org/

[11] "Intel embedded processors." Accessed: Jun. 14, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/embedded-processors.html

[12] "Spec CPU." 2006. [Online]. Available: https://www.spec.org/cpu2006

[13] "Spec CPU." 2017. [Online]. Available: https://www.spec.org/cpu2017

[14] "Synopsys dc ultra." Accessed: Jun. 14, 2024. [Online]. Available: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html

[15] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2018, pp. 131–142.

[16] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2019, pp. 399–411.

[17] M. Bekerman et al., "Correlated load-address predictors," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 1999, pp. 54–63.

[18] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2021, pp. 1121–1137.

[19] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "DSPatch: Dual spatial pattern prefetcher," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2019, pp. 531–544.

[20] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2019, pp. 1–13.

[21] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos, "Accurate and complexity-effective spatial pattern prediction," in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2004, pp. 276–287.

[22] M. Ferdman et al., "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proc. 17th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2012, pp. 37–48.

[23] M. Ferdman, S. Somogyi, and B. Falsafi, "Spatial memory streaming with rotated patterns," in *Proc. 1st JILP Data Prefetch. Championship*, 2009, pp. 1–5.

[24] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proc. 23rd Int. Conf. Supercomput.*, 2009, pp. 499–500.

[25] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2013, pp. 247–259.

[26] S. Jiang, Q. Yang, and Y. Ci, "Merging similar patterns for hardware prefetching," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2022, pp. 1012–1026.

[27] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," in *Proc. 24th Annu. Int. Symp. Comput. Archit.*, 1997, pp. 252–263.

[28] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2016, pp. 1–12.

[29] S. Kondguli and M. Huang, "Division of labor: A more effective approach to prefetching," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2018, pp. 83–95.

[30] S. Kumar and C. Wilkerson, "Exploiting spatial locality in data caches using spatial footprints," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 1998, pp. 357–368.

[31] P. Michaud, "Best-offset hardware prefetching," in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2016, pp. 469–480.

[32] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibánez, V. Viñals-Yúfera, and A. Ros, "Berti: An accurate local-delta data prefetcher," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2022, pp. 975–991.

[33] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2020, pp. 118–131.

[34] S. H. Pugsley et al., "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2014, pp. 626–637.

[35] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Multi-lookahead offset prefetching," in *Proc. 3rd Data Prefetch. Championship*, 2019, pp. 1–4.

[36] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2015, pp. 141–152.

[37] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 135–146.

[38] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2009, pp. 69–80.

[39] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2006, pp. 252–263.

[40] J. Tang, S. Liu, Z. Gu, C. Liu, and J.-L. Gaudiot, "Prefetching in embedded mobile systems can be energy-efficient," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 8–11, Jun. 2011.

[41] R. Vazquez, A. Gordon-Ross, and G. Stitt, "Energy prediction for cache tuning in embedded systems," in *Proc. Int. Conf. Comput. Design (ICCD)*, 2019, pp. 630–637.

[42] S. Volos, J. Picorel, B. Falsafi, and B. Grot, "BuMP: Bulk memory access prediction and streaming," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2014, pp. 545–557.

[43] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *Proc. Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2009, pp. 79–90.

[44] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Temporal streaming of shared memory," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2005, pp. 222–233.

[45] D. Wofk, F. Ma, T.-J. Yang, S. Karaman, and V. Sze, "Fastdepth: Fast monocular depth estimation on embedded systems," in *Proc. Int. Conf. Robot. Autom. (ICRA)*, 2019, pp. 6101–6108.

[46] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2019, pp. 996–1008.

[47] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2019, pp. 1–13.

[48] X. Zhuang and S. Pande, "Power-efficient prefetching for embedded processors," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 1, p. 3, 2007.