APAC: An Accurate and Adaptive Prefetch Framework with Concurrent Memory Access Analysis

Xiaoyang Lu, Rujia Wang, Xian-He Sun
Department of Compute Science, Illinois Institute of Technology, Chicago, IL
xlu40@hawk.iit.edu, rwang67@iit.edu, sun@iit.edu

***Abstract*—Prefetching techniques have been studied for decades. However, there are few studies on how concurrent memory accesses may affect prefetching effectiveness. When there are multiple concurrent memory requests, we can classify them into sub-classes by analyzing the overlapping relationship. In this work, we first propose *pure prefetch coverage* (PPC), a novel prefetching metric that can identify an accurate prefetch coverage under the concurrent memory access model. Then we propose APAC, an adaptive prefetch framework with PPC metric that can capture the dynamics of applications and adjust the prefetching aggressiveness. Our experimental results show that the PPC metric has a higher IPC correlation compared to the conventional prefetch coverage (PC) metric. For memory-intensive single-thread benchmarks, APAC provides an average performance improvement by 17.3% and 5.9% compared to the state-of-the-art adaptive prefetch framework FDP and NST. In a multi-core system, APAC outperforms FDP and NST by 8.5% and 5.0% IPC on average, respectively.**

I. Introduction

The unbalanced technological advancements in processor and memory over the past decades have led to the "*Memory Wall*" problem. In addition to utilizing memory hierarchy and data locality to alleviate the performance gap between the CPU and the memory, intensive research has been conducted to improve the concurrency of memory systems. Multi-port cache, multi-banked cache, and pipelined cache are advanced cache design techniques that enhance cache hit concurrency; whereas, non-blocking cache can improve cache miss concurrency. Processor ILP techniques, such as out-of-order execution, multiple issue pipeline, simultaneous multi-threading, can dramatically improve both cache hit and miss concurrency [21]. With these advanced techniques, it is common to observe concurrent memory accesses.

Memory concurrency reduces memory stall time by overlapping multiple outstanding memory accesses. Some misses occur concurrently with other hits (hit-miss overlapping), whereas some misses do not (miss-miss overlapping) [14]. Thus, a single cache miss latency is no longer a determinant factor of the overall memory system performance. The performance loss resulting from a cache miss can be reduced when there is hit-miss overlapping. When a miss has no hit-miss overlapping, it becomes the critical factor that could hurt the performance. Such miss is classified as *pure miss* (§II-A).

Data prefetching has been proved to be effective in reducing CPU stalled cycles by capturing a program's memory access pattern and then proactively fetching needed data blocks from off-chip memory to the faster on-chip cache ahead of demand access. While the conventional prefetching mechanisms are useful in reducing memory accesses delay, they are not fully utilized in a concurrent data access environment. There is room for improvement. In this work, we propose *pure prefetch coverage* (PPC), a more comprehensive metric that extends the current prefetch coverage (PC) metric to consider concurrent memory accesses. PPC evaluates the performance of a prefetcher by observing the ratio of pure misses reduced rather than misses reduced to quantify its effectiveness. We show the effectiveness of PPC compared with PC through a correlation analysis against execution time.

PPC lays a foundation for the APAC, an accurate and adaptive prefetch framework that can auto-tune the aggressiveness of the prefetcher at runtime. The memory access behavior may change phase by phase during its runtime [5]. Therefore, a prefetcher needs to be adaptive to catch the change of the data access pattern. In APAC, we measure and track the pure prefetch coverage (PPC), prefetch accuracy (PA), as well as the pure miss rate (pMR) to adjust the aggressiveness of the prefetcher dynamically. Our experimental results show that APAC outperforms state-of-the-art adaptive prefetch frameworks, such as FDP [18] and NST [9]. Also, the PPC metric can be integrated with other complex prefetchers to enhance the performance further.

The paper is organized as follows: Section II introduces the background of a concurrent memory access model and the missing piece of current prefetch metrics; Section III presents the related prefetching frameworks; Section IV introduces our proposed PPC metric and the method to measure and implement it on a given system; Section V shows our accurate and adaptive prefetch framework APAC can adjust the prefetching scheme dynamically; Section VI describes our experimental settings and Section VII presents experimental results by comparing and integrating with state-of-the-art prefetching frameworks; Section VIII concludes this paper.

II. Background And Motivation

A. Concurrent Cache Accesses

Concurrent cache accesses enable requests overlapping. Cache misses may or may not overlap with hit accesses; thus, not all misses have an equal impact on performance. When

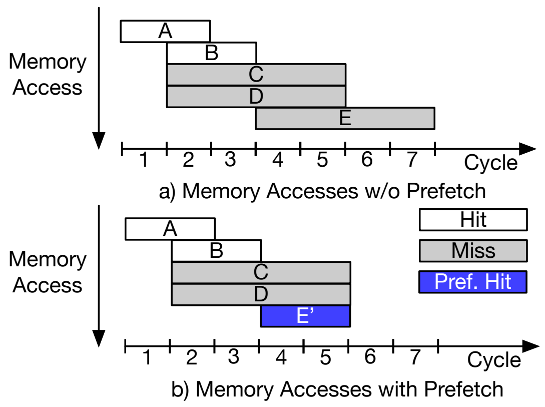

Fig. 1: Case 1: a single prefetch hit can save 2 cycles.

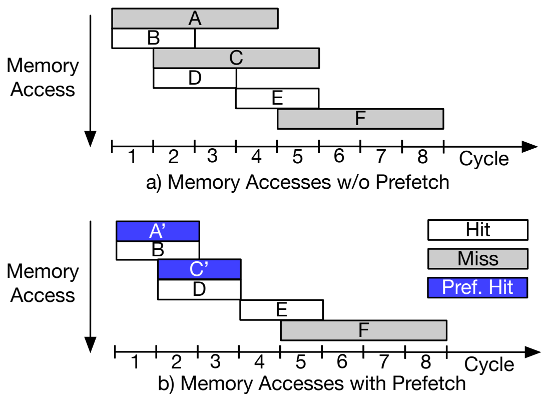

Fig. 2: Case 2: two prefetch hits do not save cycle time.

miss cycles are overlapping with one or more hit cycle, the processor can still work on the hit access(es), and the miss penalty is less significant. However, if a miss cycle has no hit to overlap with, it can severely hurt the performance. We refer a miss cycle without overlapping with any hit cycle the *pure miss cycle* and refer a miss access which consists of at least one *pure miss cycle* the *pure miss access*. In other words, *pure miss access* is the type of miss access that contains at least one miss cycle which does not have any hit accesses to overlap with [21]. We use *pure miss rate (pMR)* to define the cache efficiency when considering concurrent misses:

$$\text{Pure Miss Rate(pMR)} = \frac{\text{Num. of Pure Misses}}{\text{Num. of Total Accesses}}$$

To maximize performance, we can reduce pMR by reducing the number of pure misses via prefetching. On the other hand, we also want to minimize pure miss cycles by maximizing the hit-miss overlapping [14].

B. Prefetch Evaluation Metrics

Currently, prefetch accuracy (PA) and prefetch coverage (PC) are the most used metrics in evaluating prefetching techniques [5], [11]. PA reflects the percentage of useful prefetches out of all prefetches. Note that a useful prefetch is defined as a prefetched cache line that was accessed at least once while residing in the prefetch destination. The formal definition of prefetch accuracy is as below:

$$\text{Prefetch Accuracy} = \frac{\text{Num. of Useful Prefetches}}{\text{Num. of Total Prefetches}}$$

PC is the fraction of total misses that can be effectively reduced by prefetching [6]. Without considering concurrent memory accesses, an effective prefetcher usually means to cover as many potential misses as possible. The formal definition of prefetch coverage is:

$$\text{Prefetch Coverage} = \frac{\text{Num. of Misses Reduced by the Prefetcher}}{\text{Num. of Overall Misses w/o Prefetcher}}$$

Most prefetching techniques are designed to achieve a balanced high PA and PC. For sequential memory access activities, PC directly reflects the contribution of prefetcher to performance improvement. However, we show the limitations of the PC metric when considering concurrent memory accesses in the next section.

C. Case Studies: The Limitations of PC

PC may provide inaccurate measurements for a prefetcher when we consider concurrent data access. As discussed in Section II-A, not all misses equally impact performance, when concurrency is paramount. As a result, blindly reducing the number of misses may not be the best for performance. A high value of PC does not mean that a prefetcher can certainly cover a lot of pure misses. Likewise, if the number of cache misses saved by a prefetcher is low, but most are pure misses, a low value of PC may lead to better performance. We provide two conceptual cases in Figure 1 and Figure 2 to illustrate why ignoring concurrency information in PC metric may produce less accurate evaluations of the prefetcher's effectiveness. In both cases, each cache hit access consumes two cycles, and each cache miss has four miss penalty cycles.

Case 1: Low PC, high performance improvement. Without the help of prefetching, in Figure 1a), access A and B are cache hits, access C, D, E are cache misses. When considering the access concurrency, both access C and D have two pure miss cycles (cycle 4 and cycle 5), and access E has four pure miss cycles (cycle 4-7). According to the definition of pure miss, access C, D, and E are all pure misses. With prefetching, access E is saved by prefetch and now becomes E' in Figure 1b). Though access C and D are still misses with four miss cycles, these cycles are no longer pure miss cycles because they overlap with the hit cycles of access B and access E'. In this example, prefetching only reduces one misses, so PC is just 1/3. However, all concurrent pure misses now have hit-miss overlap. Even though the PC is relatively low, the performance gain brought by prefetching is noticeable.

Case 2: High PC, low performance improvement. The second case study shows the limitation of PC in the opposite way. In Figure 2 a), without the help of prefetching, accesses B, D, and E are cache hits, accesses A, C, F are cache misses. Access F is the only pure miss in this example, which leads to 3 pure miss cycles (cycle 6-8). All the miss penalty cycles of accesses A and C are overlapping with hits, so access A and access C are not pure misses. After prefetching, as shown in Figure 2 b), accesses A and C are saved by prefetching, and they become prefetch hits A' and C'. In this example, two misses are reduced; we calculate the PC as 2/3, which means that we saved the majority of misses. However, the total cycles spent on memory accesses are not saved. Access F is still a pure miss, with three pure miss cycles. Even though the PC is high, the prefetcher might not be able to improve performance as expected if the pure miss reduction is low.

Takeaways: The two case studies demonstrate the limitation of PC metric. When we consider memory concurrency, the correlation between the saved misses by prefetching and the memory stall cycles is loss. In some extreme cases, the correlation may even be negative. An alternative metric that considers memory concurrency is needed. Note that we do not show the performance gain of concurrent hits in Figure 1 and Figure 2 above, as they do increase hit bandwidth. Overlapping masks the data access delay of the lower layer of the memory hierarchy, which is in general significantly slower than the current memory hierarchy. The two examples show that hit-miss overlapping can directly reduce the memory stall cycles and enhance performance.

III. Related Work

In addition to studying prefetching at the algorithm level, adaptive prefetching frameworks are designed to control prefetcher aggressiveness based on runtime estimation of system performance for performance improvement.

Hur and Lin [10] introduces a probabilistic prefetching technique that utilizes stream length histograms to capture spatial locality in program execution to adjust the prefetch decision. The limitation of their framework is the lack of versatility. It cannot be adapted to other hardware prefetchers except stream prefetcher. Srinath et al. [18] design a feedback directed prefetching framework (FDP), which tracks the prefetch accuracy, prefetch lateness, and prefetcher generated cache pollution to adjust the prefetch configuration dynamically. Ebrahimi et al. [7] focus on controlling the aggressiveness of multiple prefetchers in multi-core systems based on the prefetcher-caused inter-core interference in shared memory systems. Alameldeen and Wood [3] propose an adaptive prefetching mechanism that uses cache compression's extra address tags to detect the number of useless and harmful prefetches. Near-side prefetch throttling (NST) [9] only adjusts the aggressiveness of prefetching based on the fraction of late prefetchers, which has a relatively small hardware overhead and minimizes cache pollution and memory bandwidth wastage. Although the above adjustment frameworks use different metrics as the basis for adjusting the prefetch aggressiveness, none of these metrics can properly consider access concurrency and reflect the effect of prefetch on memory performance accurately. In particular, concurrency has become the most commonly used technique in modern memory systems. As a result, these frameworks sometimes make erroneous decisions that cause the prefetcher to be too conservative or too aggressive, thereby misleading performance.

IV. Pure Prefetch Coverage

In this section, we introduce *pure prefetch coverage (PPC)*, which extends the conventional PC metric with concurrency factors. Unlike PC, PPC can examine concurrent accesses and distinguish between different types of concurrent misses. It is a comprehensive metric that evaluates a prefetcher's contribution to pure misses reduction during concurrent memory accesses. We first describe the definition and the formulas of PPC. Next, the rationality of PPC is illustrated by revisiting the two case studies. Finally, we show the algorithm and implementation details about how to track the number of pure misses and PPC during the execution time.

TABLE I: PC and PPC of two study cases

	Reduced pure misses with prefetch	Overall pure misses w/o prefetch	**PPC**	**PC**	Stall cycles reduced
Case 1	3	3	1	1/3	2
Case 2	0	1	0	2/3	0

A. Definition

The *PPC* is introduced to quantify how effective a prefetcher works in concurrent access activities. Different from the definition of PC, which relies on the ratio of total misses reduced to evaluate the effectiveness of a prefetcher, PPC focuses on quantifying the ratio of *pure misses* (§II-A) that are reduced by prefetching. PPC is defined as the fraction of the number of pure misses reduced due to prefetching over the overall number of pure misses that will occur without prefetching:

$$\text{Pure Prefetch Coverage} = \frac{\text{Num. of } \textbf{Pure Misses} \text{ Reduced by the Prefetcher}}{\text{Num. of Total } \textbf{Pure Misses} \text{ w/o Prefetcher}}$$

B. Revisit Case Studies with PPC Analysis

Recalling the two case studies in §II-C, we re-evaluated the effectiveness of prefetching using the PPC definition. Table I shows the value of PC and PPC for these two study cases, respectively. In the first case, one of the three cache misses is successfully reduced by prefetching, the value of PC is $1/3$. When considering the access concurrency, all three pure misses are now able to overlap with hits. Therefore, the value of PPC is $3/3 = 1$. A high value of PPC accurately reflects the considerable contribution of prefetching to performance gain in this case. In the second case, prefetching removes two of the three misses, so the value of PC is $2/3$. Nevertheless, pure miss is not reduced. After prefetching, access F is still a pure miss with three pure miss cycles. So the PPC is calculated as $0/1 = 0$. Compared to PC, PPC captures the ratio of pure misses reduced by prefetcher, which contributes directly to performance. PPC can accurately evaluate the effectiveness of prefetching than PC when there are concurrent memory accesses.

C. Measurement and Implementation

To compute the PPC during the execution time, we use two counters to track the pure misses: 1) RPM, is used to count the *reduced pure misses* by the prefetcher; 2) DPM, is used to record the *demand pure misses*, which are the pure misses that cannot be covered by the prefetcher. In this way, we compute PPC as:

$$\text{PPC} = \frac{\text{RPM}}{\text{RPM + DPM}}$$

where the sum of RPM and DPM is equal to the number of total pure misses without the prefetcher.

The algorithm for detecting pure misses and measuring PPC is shown in Algorithm 1. We declare the bits and counters used for measurement on the top of Algorithm 1. The information of all outstanding cache misses is tracked by the MSHR (Miss Status Holding Register). Each miss is allocated to an MSHR entry before it is served [12]. The *NoHit* and *OnlyPrefetch* bits are used to identify the current cycle status: *NoHit* is set when there is no hit in this cycle; *OnlyPrefetch* is set when there are only prefetch hits but no demand hits in this cycle.

The steps to determine whether a miss is a pure miss are shown from lines 1 to 5. A *IsPure* bit is used per MSHR entry. If *NoHit* is set, we know all misses in MSHR are pure misses, so their associated *IsPure* bits are set.

Lines 6 to 14 are used to determine how prefetch hits can reduce pure misses. If *OnlyPrefetch* is set in the cycle, these prefetch hits can be approximately considered as pure misses saved by prefetch. So we increase the RPM counter to record this type of pure miss reduction at line 8. Note that we may have multiple hit cycles, so N is divided by hit_cycle to remove repeated counts. Next, to calculate the pure misses reduced by overlapping with prefetch hits, we use an *Overlap* bit to each MSHR entry. If the *OnlyPrefetch* bit is set and a miss in the MSHR is not a pure miss, it means that it is converted from a pure miss (w/o the prefetch) to a miss that now can be overlapped with a prefetch hit. So we set its *Overlap* bit to 1 at line 11.

Lines 16 to 23 are used for updating of the counters when a miss from MSHR is serviced and removed. When a miss from MSHR entry j is serviced at this cycle, if its *IsPure* is set, the DMR counter is incremented. Otherwise, if the *Overlap* is set, the RPM counter is incremented. Then the *IsPure* and *Overlap* associated with that MSHR entry are reset. All the counters are updated at every memory cycle, so the number of demand pure misses and the number of pure misses reduced by prefetching are updated every cycle. With RPM and DPM counters, we can calculate PPC periodically based on the definition and use PPC to guide the adaptive prefetching.

Algorithm 1 Detect and Measure PPC (called every cycle)

// Single-bit cycle status identifier
NoHit: set if no hit accesses in this cycle
OnlyPrefetch: set if only has prefetch hits but no demand hits in this cycle
// Additional MSHR entry bit to record pure miss
IsPure: set if a miss is pure miss
Overlap: set if a pure miss reduced by overlapped with prefetch hits
// Counters used to compute PPC
RPM: counts the pure miss reduced by prefetching
DPM: counts the demand pure miss with prefetching

```
1:  if NoHit is set then
2:      for $ith\ outstanding\ demand\ miss\ in\ MSHR$ do
3:          $MSHR[i].IsPure = 1$
4:      end for
5:  end if
6:  if OnlyPrefetch is set then
7:      $N \Leftarrow Number\ of\ prefetch\ hits\ in\ this\ cycle$
8:      $RPM += N/hit\_cycle$
9:      for $ith\ outstanding\ demand\ miss\ in\ MSHR$ do
10:         if $MSHR[i].IsPure = 0$ then
11:             $MSHR[i].Overlap = 1$
12:         end if
13:     end for
14: end if
15:
16: for $jth\ serviced\ miss\ in\ MSHR$ do
17:     if $MSHR[j].IsPure$ is set then
18:         $DPM{+}{+}$
19:     end if
20:     if $MSHR[j].Overlap$ is set then
21:         $RPM{+}{+}$
22:     end if
23: end for
```

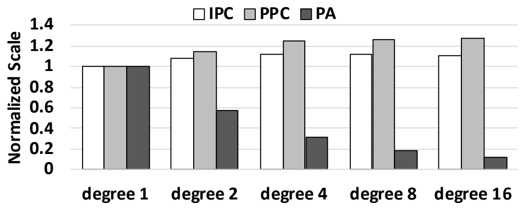

Fig. 3: The impact of aggressive prefetching on performance.

V. Adaptive Prefetch considers Access Concurrency (APAC)

Typically, a hardware prefetcher works by predicting future data access based on observed past access behavior. How aggressive a data prefetcher should be is a problem often discussed. A good data prefetcher should guarantee a sufficient aggressiveness to prefetch data ahead appropriately for the best performance. However, over-aggressive prefetching may bring adverse effects and lead to useless bandwidth consumption and cache pollution [23]. Even the memory access pattern is correctly predicted, an over-aggressive prefetcher still may create early prefetches issue [19], [22]. When the pattern prediction is inaccurate, blindly using aggressive prefetching will provide a lot of unnecessary data that evicts useful data from the cache, and could drag down the overall system performance.

Various workloads may have completely different behaviors. Even for a given workload, it may have completely different memory access patterns in different phases and show varying sensitivity to prefetch aggressiveness [13]. Additionally, the influence of concurrent data access should not be ignored if we want to evaluate the performance of systems and prefetchers correctly. To address these issues, we propose an adaptive prefetching framework APAC that takes into account data access concurrency.

A. Evaluation Metrics

In APAC, we use *pure prefetch coverage (PPC)*, *prefetch accuracy (PA)* and *pure miss rate (pMR)* without prefetching as the feedback metrics for adjusting the aggressiveness of the prefetching. The metrics are collected during each execution phase and will guide the prefetching aggressiveness in the next phase.

We first identify that using pMR can predict the overall effectiveness of the prefetcher. If a phase of an application has a high pMR, that means, in general, an aggressive prefetch algorithm is needed. In contrast, if a phase has a low pMR,

TABLE II: Adjuct prefetch aggressiveness with runtime metrics (L=Low, H=High)

Case	PPC	PA	pMR w/o Prefetch	Aggressiveness Update (reason)
1	H	H	L	No Change (base case)
2	L	H	L	Increment (to increase PPC)
3	H	L	L	Decrement (to reduce pollution)
4	L	L	L	Decrement (to reduce pollution)
5	H	H	H	Increment (to increase PPC)
6	H	L	H	No Change (to keep the high PPC)
7	L	H	H	Increment (to increase PPC)
8	L	L	H	Decrement (to reduce pollution)
PPC_{th}=0.25 PA_{low}=0.15 PA_{high}=0.4 pMR_{th}=0.5				

in most cases, we should avoid being too aggressive in prefetching to save the bandwidth and reduce pollution.

The tradeoff between prefetch aggressiveness and effectiveness can be evaluated using IPC, PPC, and PA. Figure 3 shows the behavior of the `437.leslie3d` benchmark under different aggressiveness of the IP-based stride prefetcher. The IPC, PPC, and PA all have been normalized to very conservative prefetch configuration (prefetch degree equals to 1). As the prefetch degree increases, stride prefetcher becomes more aggressive, more pure misses may effectively be covered, resulting in increased PPC. However, with the increment of the prefetch degree, useless prefetches are increasing, which is reflected in the continuous drop of PA value. In Figure 3, when the prefetch degree is 16, severe cache pollution and bandwidth contention resulted in performance degradation, yielding to lower IPC.

To achieve the optimal prefetch aggressiveness, we need to closely monitor all three metrics, PPC, PA, and pMR. We will discuss the adaptive aggressiveness selection mechanism in the next section.

B. Adaptive Aggressiveness Selection

In this work, we use the prefetch degree in the IP-based stride prefetcher to determine the prefetching aggressiveness [4]. The prefetch degree will determine how many prefetch accesses per demand miss will be generated. For example, a prefetch degree of N will bring $[A, A+1, ..., A+N]$ when there is a demand miss at address A, when the stride is 1.

We define five grades of the aggressiveness in this work (degree = 1, 2, 4, 8, 16), from the very conservative (degree 1) to the very aggressive (degree 16). The initial prefetch degree is set to 4. During the application execution time, APAC collects and evaluates the performance of prefetching at the end of each sampling phase. It dynamically adjusts the appropriate prefetching aggressiveness for the next phase based on three feedback metrics: PPC, PA, and pMR without prefetching.

The measured PPC value needs to be compared with the threshold PPC_{th} to determine whether the current aggressiveness of prefetching can cover enough pure misses. The currently measured PA value is compared with two thresholds PA_{high} and PA_{low} to determine the current prefetch accuracy is high, average or low. The threshold of pMR without prefetching pMR_{th} is used to reflect whether the current phase caused a performance issue due to the excessive number of pure misses. We set these thresholds empirically based on the results of a large number of simulations. Table II shows the thresholds used to implement APAC and the heuristic policy for dynamic updating the aggressiveness of the prefetcher.

If the value of pMR without prefetching is smaller than the pMR_{th}, except for Case 2, APAC tends to degrade the aggressiveness of the prefetcher, since we do not need to reduce the pure misses at the cost of accuracy. In Case 2, when the PA is high, and the PPC is smaller than the threshold, APAC suggests increasing the aggressiveness for higher gain from the accurate prediction. If the pMR without prefetch is larger than the pMR_{th}, the prefetcher tends to increase aggressiveness for higher PPC, which decreases the pure misses and improve the performance. Case 8 is an exception. In this case, APAC decreases the aggressiveness of prefetcher to reduce cache pollution and save memory bandwidth because the current phase shows that the prefetcher cannot prefetch effectively and accurately.

TABLE III: Hardware cost of APAC

Additional bits	Size	Used for
IsPure	1 bit per L2 MSHR	PPC, pMR
Overlap	1 bit per L2 MSHR	PPC, pMR
NoHit	1 bit	PPC, pMR
OnlyPrefetch	1 bit	PPC, pMR
DPM	32 bit	PPC, pMR
RPM	32 bit	PPC, pMR
pref-bit	1 bit per L2 block	PA, pMR
UPF	32 bit	PA, pMR
TPF	32 bit	PA

C. Hardware Cost and Complexity of APAC

APAC requires monitoring PPC, PA, and pMR without prefetching during the execution time. We have presented the measurement and implementation for tracking PPC in §IV-C. The hardware cost is shown in Table III. The needed bits include *IsPure*, *Overlap*, *NoHit*, and *OnlyPrefetch*. For a 32-entry MSHR, the *IsPure* and *Overlap* need a total of 64 bits. The *NoHit* and *OnlyPrefetch* just need 1 bit each, which is trivial. In addition, two counters, RPM and DPM are required. The 32-bit wide registers are sufficient to prevent their data overflow.

To measure PA value, we use a similar method described by Feedback Detected Prefetching (FDP) [18]. A bit *pref-bit* per L2 block is required to differentiate whether the data block comes from demand request or prefetch request. For a 256KB L2 cache with 64B cache block size, the total *pref-bit* size is 0.5KB. With the help of *pref-bit*, the number of useful prefetches (prefetch hits) and the total number of prefetches can be recorded by two 32-bit wide counters UPF and TPF. The value of PA can be calculated as the ratio between the UPF and TPF.

The pMR value without prefetching is computed as the ratio of the total number of pure misses that will occur without prefetching to the number of total accesses that will occur without prefetching. We use the method mentioned in §IV-C to count the number of pure misses that will occur without prefetching. The number of accesses that will occur without prefetching can be obtained by calculating the sum of demand

TABLE IV: Simulated system configurations

Processor	One to four cores, 4GHz, 8-issue width, 256-entry ROB
L1 Cache	split 32KB I/D-cache/core, 8-way, 4-cycle hit latency, 8-entry MSHR, 64B line-size
L2 Cache	unified 256KB, 8-way, 10-cycle hit latency, 32-entry MSHR, 64B line-size
L3 Cache	shared 2MB/core, 16-way, 20-cycle hit latency, 64×#cores-entry MSHR, 64B line-size
DRAM	4GB 1 channel, 64-bit channel, 1600MT/s

misses, demand hits, and prefetch hits. Therefore, the bits and counters used for PPC and PA can help to compute pMR without prefetching as well.

In total, the hardware overhead of APAC is around 0.52 KB, which is only 0.2% of the capacity of the baseline 256KB L2 cache.

VI. Experimental Methodology

We implement our adaptive prefetching framework APAC as described in §V with both a single-core system and a 4-core system. ChampSim [1] simulator is used to provide an appropriate memory system performance simulation. A detailed out-of-order CPU model in the ChampSim was adopted to achieve the most accurate simulation results. The details of the configuration parameters of our simulation are described in Table IV. APAC works with an IP-based stride prefetcher at the L2 cache by default. As we mentioned in §V-B, there are five different prefetch degrees (1,2,4,8,16) available for selection by the stride prefetcher.

For APAC dynamic prefetching framework, the value of PPC, PA, and pMR without prefetching will be updated every 4096 misses (half the number of blocks in the L2 cache) in L2. Initially, APAC will set the prefetch aggressiveness of the first phase to degree 4. When a given stride prefetcher works under the APAC frame, the prefetch degree will be dynamically adjusted between degree 1 and degree 16, and the prefetcher will never be disabled.

We select the performance without a prefetcher as the baseline for performance comparison. We compare APAC against two state-of-the-art adaptive prefetching frameworks FDP [18] and NST [9]. We also implement a naive adaptive framework called NAP with a similar workflow as APAC. However, NAP makes all decisions without considering concurrency. NAP dynamically adjusts the aggressiveness of prefetching based on prefetch coverage (PC), prefetch accuracy (PA), and miss rate (MR) without prefetching. The importance of comprehensive memory access analysis can be reflected by comparing APAC and NAP.

We collect SimPoint [16] traces from SPEC CPU2006 [17] and SPEC CPU2017 [2]. For SPEC workloads, we use high intensity workloads with MPKI > 3, as shown in Table V. For 4-core experiments, we test multi-copy and mixed SPEC workloads. A multi-copy workload has four identical copies of a single benchmark. A mixed workload has four different benchmarks, which are assigned to different cores. CloudSuite [8] workloads are multi-threaded and are only used for 4-core experiments. Each trace is warmed up with 50M instructions

TABLE V: Evaluated workloads

Workload	LLC MPKI	Workload	LLC MPKI
436.cactusADM	4.99	437.leslie3d	3.56
459.GemsFDTD	6.40	462.libquantum	26.07
482.sphinx3	11.65	602.gcc	70.06
603.bwaves	23.19	605.mcf	72.69
619.lbm	47.23	620.omnetpp	10.64
621.wrf	19.22	623.xalancbmk	19.10
649.fotonik3d	8.77	654.roms	32.47
MIX1	436,437,462,482	MIX2	436,437,602,603
MIX3	436,437,621,623	MIX4	436,437,649,654
MIX5	462,482,602,603	MIX6	462,482,621,623
MIX7	462,482,649,654	MIX8	602,603,621,623
MIX9	602,603,649,654	MIX10	621,623,649,654

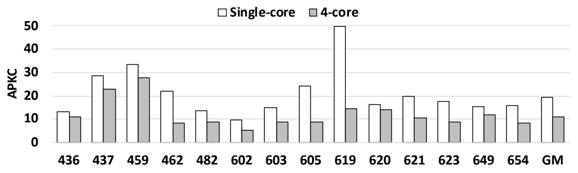

Fig. 4: APKC on L2 in single-core and 4-core configurations.

for all experiments, and simulation results are collected over the next 200M instructions.

VII. Experimental Results

In this section, we first discuss the concurrency of each SPEC workloads, then verify the correctness of PPC through a performance-metric correlation study. Finally, we show the effectiveness of the APAC.

A. Concurrency Analysis

We use *accesses per kilocycles (APKC)* [20] to measure the overall memory concurrency concerning the complexity of modern memory systems. Figure 4 shows the L2 concurrency of each SPEC benchmark in single-core and 4-core multi-copy configurations without prefetching. In the multi-core system, applications run on different cores and share the LLC and main memory, which causes bandwidth contention, especially when all cores are running the same application. This results in the concurrency gap between the single-core and 4-core multi-copy configurations shown in Figure 4. The single-core cases achieve a geometric mean of 1.9 times higher APKC than 4-core cases.

B. Accuracy of PPC Metric

We show the correlation between PPC and IPC for each evaluated workload to verify the correctness of the PPC metric. Also, we show the correlation between the classical metric PC and IPC for comparison. The higher the correlation is, the better the metric is, whereas a low correlation means the metric is wrong. From a statistical point of view, the correlation coefficient describes the proximity between the changing trends of the two variables. Therefore for each application, five static prefetch configurations(from degree 1 to degree 16) are executed independently. Then, we calculate the $r(IPC, PPC)$ and $r(IPC, PC)$ correlation based on the five different configurations. The correlation coefficient of r of two variables X and Y can be calculated using the following equation:

TABLE VI: Performance correlation coefficient analysis

Workload	single-core		4-core	
	r(PPC,IPC)	r(PC,IPC)	r(PPC,IPC)	r(PC,IPC)
436.cactusADM	0.97	-0.65	0.99	0.78
437.leslie3d	0.98	0.89	0.99	0.70
459.GemsFDTD	0.99	0.40	0.78	0.39
462.libquantum	0.96	0.92	0.99	0.95
482.sphinx3	0.99	0.89	0.99	0.89
602.gcc	0.98	0.86	0.70	0.21
603.bwaves	0.91	0.52	0.98	0.89
605.mcf	0.98	0.80	0.99	0.89
619.lbm	0.83	0.56	0.65	0.34
620.omnetpp	0.96	0.87	0.66	0.28
621.wrf	0.99	0.94	0.94	0.84
623.xalancbmk	0.99	-0.55	0.85	0.75
649.fotonik3d	0.99	0.95	0.95	0.91
654.roms	0.99	0.95	0.99	0.91
Average	0.97	0.60	0.89	0.70

$$r(X,Y)=\frac{n(\sum XY)-(\sum X)(\sum Y)}{\sqrt{[n\sum X^2-(\sum X)^2][n\sum Y^2-(\sum Y)^2]}}$$

where X and Y are the sampling points for two variables.

Table VI shows that, in both single-core and 4-core configurations, compare to PC, PPC shows a stronger positive correlation with IPC. This result demonstrates the unique advantage of PPC in capturing the concurrency characteristics of modern memory systems and accurately evaluating the efficiency of prefetching. As discussed in multi-core cases, the concurrency of memory accesses will be reduced by bandwidth contention. The accuracy of PPC will be affected by concurrency, which will cause the average gap between $r(IPC, PPC)$ and $r(IPC, PC)$ in 4-core configurations to be smaller than the gap in single-core configurations.

C. APAC Performance Evaluation

Single-core Results: Figure 5 shows the single-core pure prefetch coverage of the various adaptive prefetch frameworks. APAC achieves the highest PPC of all the adaptive prefetch frameworks simulated. APAC covers 40.0% of the demand pure misses at L2, higher than NST's 35.7%, FDP's 29.0%, and NAP's 22.2%. For `459.GemsFDTD` and `620.omnetpp`, which suffer from indirect accesses, APAC does not show advantages on PPC. The hardware prefetchers do not prove to be useful to these benchmarks with irregular accesses, and all frameworks are almost not conducive to the performance with a less than 0.05 pure miss coverage at the L2.

Figure 6 shows the single-core speedup achieved by NAP, FDP [18], NST [9] and APAC for the individual memory-intensive SPEC CPU applications, followed by the geomean across all the workloads. All results are normalized to the baseline of no prefetching. In most cases, APAC can match the best static optimum for each specific workload based on feedbacked application phase behavior. APAC provides a 70.0% higher geometric mean IPC over the baseline, 28.9% over NAP, 17.3% over FDP, and 5.9% over NST. Benchmarks `603.bwaves`, `619.lbm`, `621.wrf`, and `623.xalancbmk`, benefit the most from APAC, the IPC increased over NST ranging from 8.6% to 49.2%. For these benchmarks, based on the observation of PPC, APAC provides

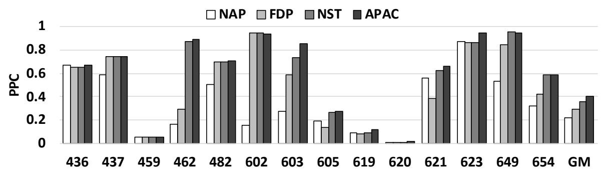

Fig. 5: PPC measured in the single-core configuration.

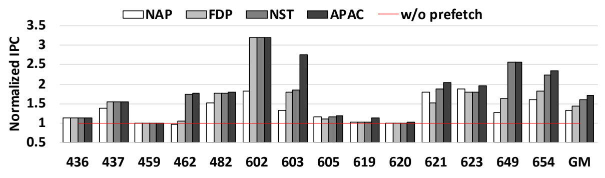

Fig. 6: Normalized IPC compared to baseline (single-core).

higher PPC over NST, ranging from 5.3% to 34.0%, which can explain why these benchmarks benefit the most from APAC. APAC fully considers the balance between the reduction of pure misses and the accuracy of the prefetch requests. Therefore APAC can prevent severe cache pollution while ensuring pure miss coverage.

4-core Results: For the evaluation of 4-core systems, we simulate both multi-copy and mixed workloads then compare APAC with other adaptive mechanisms. For multi-copy workloads, Figure 7 shows that APAC provides superior performance improvement with 12.1% higher geometric mean over the baseline, whereas both FDP and NST only provide 6.6% speedup. Compared with single-core results, the effectiveness of all adaptive prefetch frameworks has decreased. The contention at the LLC and DRAM bandwidth is the primary limiting factor to cause this trend.

For mixed workloads, Figure 8 shows that APAC achieves an improvement of 62.7% on average, whereas NAP, FDP, and NST improve performance by 28.6%, 40.3%, and 53.1%. In the multi-core system, coordinated throttling is independently applied to the L2 prefetcher of each core, which is essential for mixed workloads with different access patterns and non-uniform bandwidth demands.

For most CloudSuite benchmarks, it is challenging for most hardware prefetchers to capture their complex access patterns. Since the focus of APAC is not to detect and propose complex prefetching strategy, we achieve similar performance gains compared with other frameworks. As shown in Figure 9, `streaming` is the only benchmark where all frameworks work can significantly improve performance. On average, APAC achieves a 10.6% speedup and outperforms NAP by 3.6%.

Integrate APAC with a complex prefetcher: The major contribution of this paper is a framework that enables comprehensive concurrent access pattern analysis, and we have shown that with a simple strided prefetcher, we can enhance the performance for most workloads. It is worth noting that our approaches can be easily integrated with more complex prefetching algorithms and extended through multiple memory hierarchies. By adequately integrating our PPC metrics into the system, the performance gain coming from the advanced prefetching algorithms can be further enhanced with our meth-

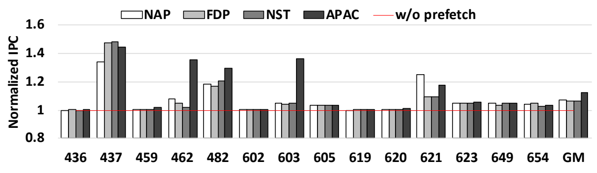

Fig. 7: Normalized IPC compared to baseline (4-core, multi-copy).

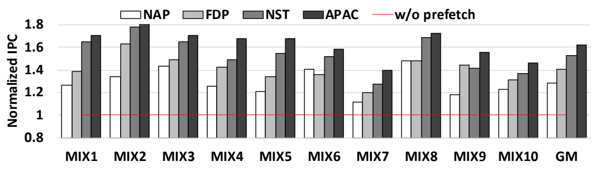

Fig. 8: Normalized IPC compared to baseline (4-core, mixed-copy).

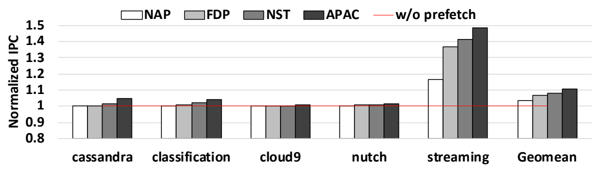

Fig. 9: Normalized IPC compared to baseline (4-core, Cloud-Suite).

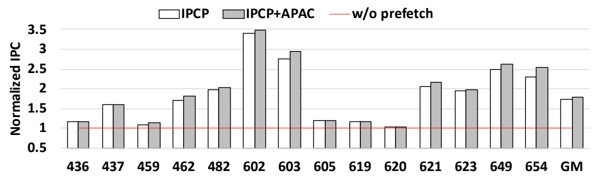

Fig. 10: Speedup of APAC + IPCP in the single-core configuration.

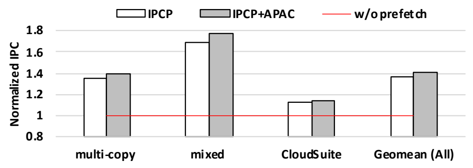

Fig. 11: Speedup of APAC + IPCP in the 4-core configuration.

ods. We apply APAC to the open-sourced IPCP [15], which is the winner of the 3rd Data Prefetching Championship (DPC-3). IPCP can perform multiple types of prefetching patterns; therefore, prefetching accuracy is relatively high. We show the add-on performance gain in Figure 10 and Figure 11. Compare to utilizing IPCP alone, applying APAC to the IPCP provides additional performance improvement of 3.2% and 3.4% in the single-core and 4-core configuration, respectively.

VIII. Conclusions

In this paper, we identify that concurrency of memory accesses is an indispensable factor when evaluating the prefetch effectiveness. We propose *pure prefetch coverage* (PPC), a comprehensive metric focusing on the effect of prefetching. We develop a detailed implementation of detecting pure misses and the measurement method for PPC. Furthermore, we design an accurate and lightweight, adaptive prefetch framework, *APAC*, based on concurrency aware metrics. APAC outperforms state-of-the-art adaptive prefetcher frameworks, and it can be easily integrated with other advanced prefetchers.

Acknowledgment

This research is supported in part by the National Science Foundation under grant CCF-2008907, CNS-1730488 and CCF-1536079, and by the NSF Chameleon computing facility.

References

[1] The champsim simulator. https://github.com/ChampSim/ChampSim.
[2] Spec cpu2017 benchmark suite. http://www.spec.org/cpu2017/.
[3] A. R. Alameldeen and D. A. Wood. Interactions between compression and prefetching in chip multiprocessors. In *HPCA*, 2007.
[4] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *SC*, 1991.
[5] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez. Perceptron-based prefetch filtering. In *ISCA*, 2019.
[6] Y. Chen, H. Zhu, and X.-H. Sun. An adaptive data prefetcher for high-performance processors. In *CCGRID*, 2010.
[7] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *MICRO*, 2009.
[8] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Acm sigplan notices*, 47(4):37–48, 2012.
[9] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur. Near-side prefetch throttling: adaptive prefetching for high-performance many-core processors. In *PACT*, 2018.
[10] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *MICRO*, 2006.
[11] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching. In *MICRO*, 2016.
[12] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA*, 1998.
[13] J. Lee, H. Kim, and R. Vuduc. When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(1):1–29, 2012.
[14] Y. Liu and X.-H. Sun. Lpm: A systematic methodology for concurrent data access pattern optimization from a matching perspective. *IEEE Transactions on Parallel and Distributed Systems*, 30(11):2478–2493, 2019.
[15] S. Pakalapati and B. Panda. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *ISCA*, 2020.
[16] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and efficient simulation. *ACM SIGMETRICS Performance Evaluation Review*, 31(1):318–319, 2003.
[17] C. D. Spradling. Spec cpu2006 benchmark tools. *ACM SIGARCH Computer Architecture News*, 35(1):130–134, 2007.
[18] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *ISCA*, 2007.
[19] V. Srinivasan, E. S. Davidson, and G. S. Tyson. A prefetch taxonomy. *IEEE Transactions on Computers*, 53(2):126–140, 2004.
[20] X.-H. Sun and D. Wang. Apc: a performance metric of memory systems. *ACM SIGMETRICS Performance Evaluation Review*, 40(2):125–130, 2012.
[21] X.-H. Sun and D. Wang. Concurrent average memory access time. *Computer*, 47(5):74–80, 2013.
[22] S. P. Vanderwiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys (CSUR)*, 32(2):174–199, 2000.
[23] X. Zhuang and S. L. Hsien-Hsin. Reducing cache pollution via dynamic data prefetch filtering. *IEEE Transactions on Computers*, 56(1):18–31, 2006.