Lookahead Prefetching with Signature Path

Jinchun Kim, Paul V. Gratz and A. L. Narasimha Reddy
Electrical and Computer Engineering, Texas A& M University
cienlux@tamu.edu, pgratz@gratz1.com, reddy@tamu.edu

***Abstract*—Existing data prefetchers speculate on spatial and temporal locality by tracking past memory accesses. Relying on the past memory accesses restricts the scope of prefetching and potentially further performance improvement. In this paper, we propose a lookahead prefetching algorithm called Signature Path Prefetching (SPP) that accurately predicts the next memory access pattern and exploits this future access to initiate lookahead prefetching. Unlike prior lookahead algorithms, SPP is purely based on the memory access stream and does not require additional support from branch history, PC, or metadata to lookahead future memory access. Within a 32KB storage limit, we evaluate SPP under different memory constrained scenarios and find SPP outperforms the previous competition winner AMPM prefetcher by 4% performance improvement.**

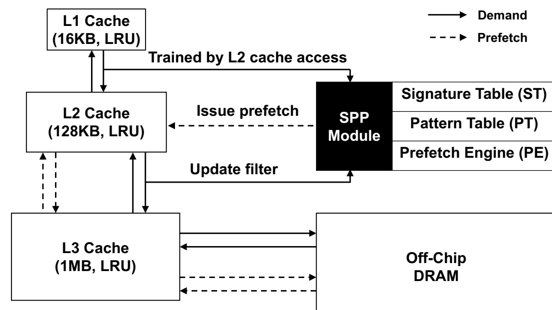


Figure 1: Overall SPP architecture

I. Introduction

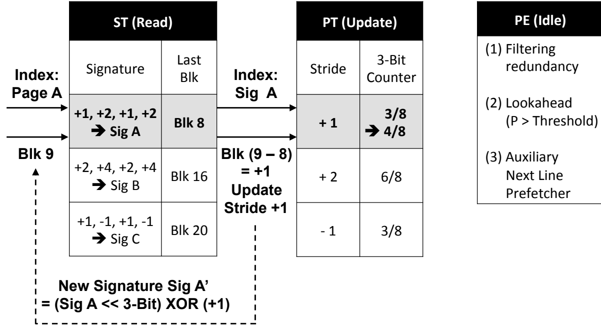
Data prefetching can provide an efficient means to improve the performance of modern microprocessors. The aim of the technique is to proactively fetch useful data blocks from long-latency off-chip DRAM to the faster on-chip SRAM cache ahead of demand access. Typically, prefetching techniques predict the future access pattern based on past memory accesses. Thus, prefetching hardware speculates on spatial and temporal locality based upon learning of past program behavior. In these traditional prefetching techniques, prediction is inherently limited by the number of past access patterns that have been monitored. Moreover, prefetching should be very accurate. Even if there is a prefetcher that can hold limitless amount of information, prefetched data will pollute the cache if data is unused or untimely. Therefore, it is highly desirable to develop a prefetching algorithm that can predict many future accesses with high accuracy.

To address both prefetching scope and accuracy, prior works adopted lookahead mechanism into data prefetching [2], [6], [7]. Previous studies, however, suffer from high hardware complexity and require additional support from the core pipeline. For example, B-Fetch requires branch history and a copy of architectural register file to perform lookahead prefetching [2]. Although this extra information shows potential to improve performance, exporting this information down to the lower level caches implies implementation challenges as this information is not typically required in low-level caches [3].

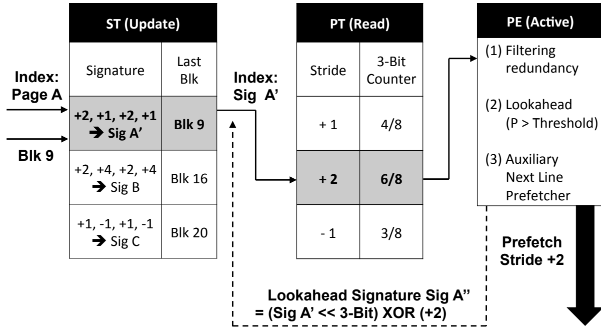
In this work, we propose a simple but powerful lookahead prefetching algorithm called Signature Path Prefetching (SPP) that aggressively speculates beyond the current demand memory access and traverses down the future memory access pattern that is likely to be used. Unlike prior lookahead prefetchers [2], [6], [7], SPP does not require additional support from branch information, PC, or cache metadata and is purely based on the memory access stream. Since there are no hooks between core pipeline and SPP, the prefetching engine can operate as a stand alone module without the complexities of exporting core information. We evaluate SPP with 16 different SPEC CPU 2006 benchmarks and achieve 26.1% performance improvement compared to a processor without prefetching. Moreover, SPP outperforms AMPM prefetcher [1], the winner of the previous data prefetching competition by 4% on average.

II. Design

The high level design of the SPP engine is illustrated in Figure 1. The SPP module is a three-stage, pipelined structure that consists of a Signature Table (ST) stage, a Pattern Table (PT) stage, and the Prefetch Engine (PE). SPP is trained by L2 cache accesses (L1 Misses) and issues prefetch requests into the L2 read queue. The ST stage is indexed by physical page number (PPN) and stores the previously seen memory access pattern as a compressed 12-bit signature. The PT stage is indexed by a history signature generated from the ST stage and stores future access stride patterns. The PT stage also estimates the probability that a given access stride pattern will yield a useful prefetch. If the stride in the PT is found to have sufficient probability (above a configured threshold), this pattern is passed to the PE for prefetch generation. As noted in Figure 1, prefetching puts additional pressure on both cache and DRAM accesses. Therefore, it is important to detect redundant prefetching requests and filter them out properly. To avoid unnecessary prefetching requests, we implement a filter at the PE stage.



(a) Read previous signature (*Sig A*) from matching page (*Page A*) and update PT with current stride (*+1*)



(b) Update new signature (*SigA'*) and predict next stride

Figure 2: SPP design and operations

The filter excludes prefetch requests from the PT stage that are already in flight or have been previously demanded. We use L2 demand, prefetch, fill, and eviction information to update this filter. In the remainder of this section we describe each stage of the SPP in detail.

A. Signature Table Stage

The ST is designed to capture a memory access pattern within a 4KB physical page and to compress previous strides into a 12-bit history signature. Figure 2 shows an example of the SPP including the ST stage. To capture local access patterns in a 4KB spatial region, the ST is indexed with a hash of the PPN, tagged with the lower 8-bits of the PPN. For simplicity, the ST is illustrated as a direct map structure in Figure 2, in fact it is organized as a 2-way associative cache with the PPN hash acting as an index to the set. The table also stores the last block accessed in a page to calculate current stride and update the PT.

Whenever there is a L2 cache access, the physical address of this access is passed into ST to find a matching entry for this physical page. Figure 2a shows an example of accessing the ST with *Page A* and block offset *Blk 9*. In this case, the ST finds matching entry for this page and is able to provide a stored signature (*Sig A*). This signature is a compressed representation of a previous access patterns to the given page, which is generated via a series of XORs and shifts as we discuss in the next paragraph. In this case, *Sig A* represents a previous access pattern to *Page A* which was *(+1,+2,+1,+2)*. Since the ST also stores the last block offset (*Blk 8*) accessed in *Page A*, we know that current stride in *Page A* is *Blk (9 - 8) = (+1)*. This stride is non-speculative since it is based on a demand request from the L1 to the L2. Therefore, we can infer that a given set of accesses (*Sig A* here) will lead to a next stride of *(+1)*. This correlation is stored in the PT as shown in Figure 2a.

$$\textit{New Signature} \\ = (\textit{Old Signature} \ll \textit{3-Bit}) \text{ XOR } (\textit{Current Stride}) \qquad (1)$$

After updating the PT, the ST is also updated with a new signature based on current stride *(+1)*. Equation 1 shows how SPP generates a new history signature. The old signature (*Sig A*) is left shifted 3-bits and XOR'ed with the current stride *(+1)*. In this way, the 12-bit signature can represent the last four memory accesses in *Page A*. We note that we refer to this signature as being compressed because stride deltas of greater than 7 have the potential to overlap. At this point, the new signature (*Sig A'*) represents a current access pattern to *Page A* of *(+2,+1,+2,+1)*. Since *Sig A'* represents current access pattern, SPP searches for *Sig A'* in the PT so that it can predict the next pattern following *Sig A'*. Figure 2b shows that *Sig A'* is updated in the ST and *Sig A'* finds a possible prefetch candidate of stride *(+2)* in the PT. The last block offset is also updated from *Blk 8* to *Blk 9* in ST.

If there is no matching entry in the ST, it means that current access pattern has not been seen before for this page or that this page has been evicted from the ST. In both cases, the ST is not able to provide a new signature since there is no accumulated history for this page. One possible option is to set the initial signature as zero. However, setting an initial signature to zero causes substantial conflicts in PT since every ST miss will access to the set zero of PT and pollutes the stride pattern. To avoid such pollution, we use current block offset as an initial signature when ST miss occurs so that we can properly distribute initial signatures to PT.

B. Pattern Table Stage

As described in the previous section, the PT holds the potential next stride patterns that correspond to specific history signature. Therefore, the PT is indexed by the signature and each set contains the predicted next stride. Unlike the ST, whose entries correspond to each physical page, each PT entry can be shared regardless of page number. This is possible since multiple pages can show same memory access pattern. In other words, if *Page A* and *Page B* show the same access pattern, they will generate same signature, index to same entry in the PT, and update same stride pattern in PT. In doing so, the globally shared PT accelerates the learning process of SPP. The PT is designed as a 4-way set associative structure so that multiple strides can be prefetched by a

single signature. Each stride in the PT can be prefetched if a corresponding counter is above a given prefetch threshold. For simplicity, the PT is illustrated as a direct map structure in Figure 2.

Figure 2a shows how the PT is updated with a history signature. Since a matching stride *(+1)* that corresponds to *Sig A* is found in PT, the 3-bit counter increases by one. Once the counter value becomes greater than prefetching threshold, this stride is considered as a prefetch candidate. If there is no matching stride, PT replaces the entry with the lowest counter value and decreases all other counters in that set by one. Decreasing the counter on a PT miss is particularly effective when a process shows a random access pattern, since it allows SPP to automatically throttle inaccurate prefetching.

As shown in Figure 2b, matching strides above the prefetch threshold are marked as prefetch candidates and delivered to the PE stage. We set the prefetch threshold to 50% in the final design. In addition, the PT provides a lookahead candidate to the PE stage for lookahead mechanism. A lookahead candidate is selected among prefetch candidates whose counter value probability is above 75% lookahead threshold. Since there could be multiple strides that exceed 75% lookahead threshold, SPP selects only one lookahead candidate with the maximum counter value.

C. The Prefetch Engine

The main objective of the PE is to issue prefetches and activate the lookahead mechanism. Figure 2b explains how prefetching and lookahead is handled in the PE. First, the PE always prioritizes issuing prefetch candidates over performing lookahead since the lookahead process represents a deeper level of speculation. To drop redundant prefetch candidates that have already been demanded or prefetched, we implement a filter that records a 64-bit bitmap vector per page. Since there are 64 cache blocks in a page, a single 64-bit vector can cover the 4KB spatial region. Each bit in a vector is set to 1 when corresponding cache block is demanded, prefetched or filled. If the filter detects a prefetch request that is already set to 1, this prefetch request will be ignored and PE does not issue prefetch. The bitmap is reset to 0 when the cache block is evicted from L2 cache. After checking the redundancy filter, SPP issues prefetch request (stride delta *(+2)*) and the request is placed on L2 read queue.

The PE initiates the lookahead process by building a speculative *lookahead signature*. As shown in Figure 2b, the lookahead signature (*Sig A"*) is generated from *Sig A'* and predicted stride *(+2)* using same Equation 1. The lookahead signature is used to index the PT again such that SPP can search for further prefetch and lookahead candidates down the signature path. If *Sig A"* finds more prefetch candidates in the PT, the aforementioned filtering process will be repeated and proper prefetch candidates will be issued. Moreover, if *Sig A"* finds another lookahead candidate, the lookahead mechanism will be performed recursively until the signature path confidence falls below the lookahead threshold.

Although *Sig A"* results from the same Equation 1, *Sig A"* always has lower confidence than *Sig A'*. This is because *Sig A'* is based on *non-speculative* current stride *(+1)* while *Sig A"* is based on *speculative* stride *(+2)*. Also, counter value used to identify the lookahead candidate only provides the *probability* of using a specific stride and does not guarantee the *confidence* of signature path for lookahead process [5]. To emulate such path confidence with counter value, we keep the cumulative path probability by multiplying prior lookahead probabilities as SPP traverses down the signature path. If the cumulative path probability falls below the lookahead threshold (75%), indicating a likelihood of wrong signature path prediction, the lookahead process is terminated. To naturally throttle the aggressive lookahead process, SPP sets the maximum lookahead probability to 95%. The PE also throttles prefetch and lookahead based on the remaining L2 cache bandwidth. Reserving L2 cache bandwidth is desirable to avoid traffic congestion because SPP puts prefetch requests on the L2 read queue. Therefore, the PE does not issue prefetch or lookahead further if more than half of L2 read queue is occupied. The auxiliary next line prefetcher gets activated only when L2 cache read queue is not congested and there is no prefetch candidate.

III. EVALUATION

We evaluate SPP with 16 different SPEC CPU2006 benchmarks. Each benchmark is fastforwarded with different number of instructions according to the methodology introduced by Khan *et al.* [4]. All experiment results are warmed up with 100M instructions and simulated for additional 500M instructions. SPP requires 30.94KB storage which satisfies the 32KB storage limit for the prefetching competition. Table I lists the storage breakdown.

Structure	Components			Number of Bits		Storage	
Signature Table	512 Sets	2-Way	Valid	1	$= 512 \times 2 \times 1$	27648	**239616 Bits = 30.94 KB**
			Tag	8	$= 512 \times 2 \times 8$		
			Signature	12	$= 512 \times 2 \times 12$		
			Last Block	6	$= 512 \times 2 \times 6$		
Pattern Table	4096 Sets	4-Way	Valid	1	$= 4096 \times 4 \times 1$	188416	
			Stride	7	$= 4096 \times 4 \times 7$		
			Counter	3	$= 4096 \times 4 \times 3$		
		Lookahead Candidate		2	$= 4096 \times 2$		
Prefetch Engine (Filter)	256 Sets	2-Way	Valid	1	$= 256 \times 2 \times 1$	37376	
			Tag	8	$= 256 \times 2 \times 8$		
			Bitmap	64	$= 256 \times 2 \times 64$		

Table I: SPP storage computation

Figure 3 shows the IPC improvement results over the baseline processor without prefetching. We compare SPP with four different prefetching techniques: Stream, IP Stride, Next Line, and AMPM. Overall, SPP outperforms these

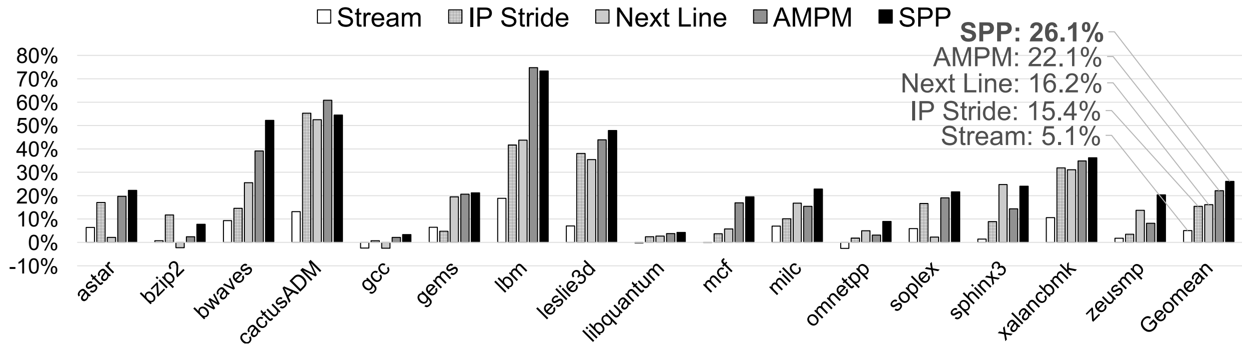


Figure 3: IPC improvement results (1MB L3 Cache and 12.8 GB/s DRAM bandwidth)

prefetchers or shows equivalent performance from most of benchmarks except bzip2 and cactusADM. We found that bzip2 mostly benefits from learning prefetch patterns based on the instruction pointer (IP stride). Meanwhile, AMPM outperforms SPP on cactusADM by 6.3%. We found that this is due to a particularly noisy stride pattern in cactusADM which pollutes the history signature stored in the ST. Experimentally, we found that the impact of the noisy stride can be minimized by generating a longer history signature. We confirm that extending signature length from 12-bit to 13-bit allows SPP to catch up the performance of AMPM for cactusADM. In addition, longer signature also improves the performance of other benchmarks. However, due to the 32KB storage limit, we restrict the signature length to 12-bit. In geometric mean, SPP shows 26.1% IPC improvement while AMPM shows 22.1%. Since the performance of Stream, IP Stride, and Next Line do not approach that of AMPM, we focus on comparing SPP and AMPM from here on.

Figure 4 shows the geometric mean performance improvement from different memory configurations. The baseline configuration is same as Figure 3 which has 1MB L3 cache and 12.8 GB/s DRAM bandwidth. The Small LLC configuration changes the size of the L3 cache to 256KB. With a smaller L3 cache, AMPM experiences 1.5% performance degradation while SPP shows only a 0.5% degradation versus baseline. Due to the timely prefetching based on the history signature, SPP utilizes the small L3 cache better than AMPM. Low Bandwidth changes the DRAM bandwidth of the system to 3.2 GB/s. Neither prefetching technique performing particularly well due to the restricted bandwidth. However, SPP still exceeds the performance of AMPM by 1.1%. Scramble Loads does not change the size of cache or DRAM bandwidth but this configuration puts additional randomness to the L1 access stream. In this configuration, both prefetchers see a minor performance degradation. The randomization impacts SPP more than AMPM since the history signature is based on the sequence of access strides. Still, SPP outperforms AMPM by 3.5% with the randomization.

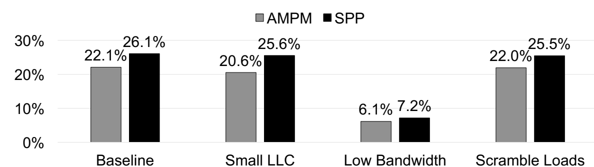


Figure 4: Geometric mean IPC improvement from different memory configurations

IV. Conclusions

Lookahead prefetching is an attractive way to improve traditional prefetching algorithms since it can capture more prefetch candidates in advance. In this work, we present SPP which does not require additional information from core processor to perform lookahead prefetching. Our design shows 26.1% performance improvement which exceeds the performance of previous competition winner AMPM by 4%.

References

[1] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," *Journal of Instruction-Level Parallelism*, vol. 13, 2011.

[2] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. V. Gratz, and D. A. Jiménez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[3] S. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutlu, and D. A. Jiménez, "Improving cache performance by exploiting read-write disparity," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[4] S. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in *Proceedings of the 43th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[5] K. Malik, M. Agarwal, V. Dhar, and M. I. Frank, "Paco: Probability-based path confidence prediction," in *Proceedings of the 14th International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[6] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA)*, 2003, pp. 129–140.

[7] S. S. Pinter and A. Yoaz, "Tango: a hardware-based data prefetching technique for superscalar processors," in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, 1996.