S/DC: A Storage and Energy Efficient Data Prefetcher

Xianglei Dang, Xiaoyin Wang, Dong Tong, Junlin Lu, Jiangfang Yi, Keyi Wang
Microprocessor Research & Development Center, Peking University, Beijing, China
{dangxianglei, wangxiaoyin, tongdong, lujunlin, yijiangfang, wangkeyi}@mprc.pku.edu.cn

Abstract—**Energy efficiency is becoming a major constraint in processor designs. Every component of the processor should be reconsidered to reduce wasted energy and area. Prefetching is an important technique for tolerating memory latency. Prefetcher designs have important impact on the energy efficiency of the memory hierarchy. Stride prefetchers require little storage, but cannot handle irregular access patterns. Delta correlation (DC) prefetchers can handle complicated access patterns, but waste storage because of storing multiple miss addresses for a stride pattern. Moreover, DC prefetchers waste the bandwidth and energy of the memory hierarchy because they cannot identify whether an address has been prefetched and generate a large number of redundant prefetches.**

In this paper, we propose a storage and energy efficient data prefetcher called stride/DC (S/DC) to combine the advantages of stride and DC prefetchers. S/DC uses a pattern prediction table (PPT) which stores two recent miss addresses in each entry to capture stride patterns. PPT avoids recording multiple miss addresses for a stride pattern, and thus improves the storage efficiency. When handling stride patterns, each PPT entry maintains a counter for obtaining the last prefetched address to avoid generating redundant prefetches. When handling other patterns, S/DC compares the new predicted address with earlier generated addresses in the prefetch queue and filters the redundant ones. In addition, to expand the filtering scope, S/DC uses a prefetch filter to store addresses evicted from the prefetch queue. In this way, S/DC reduces the bandwidth requirements and energy consumption of prefetching. Experimental results demonstrate that S/DC achieves comparable performance with only 24% of the storage and reduces 11.46% of the L2 cache energy, as compared to the CZone/DC prefetcher.

I. Introduction

With the growing concerns about power and complexity, energy efficiency is becoming a key constraint in processor designs [1][2]. To optimize energy efficiency, all components of the processor should be reconsidered to reduce wasted energy and area. Prefetching [3][4] has been widely used in processors to tolerate memory latency. Prefetcher designs have important impact on the performance and energy of the memory hierarchy [5]. The storage occupied by the prefetcher to record history data incurs area and power consumption. On the other hand, each cache miss will access the prefetcher and potentially invokes multiple prefetches. Prefetch addresses from consecutive misses may overlap and incur redundant prefetches, which waste bandwidth and energy [6]. Therefore, it is critical to reduce the occupied storage and redundant prefetches in order to improve the energy efficiency.

This work was supported by the National High Technology Research and Development Program of China (No. 2009ZX01029-001-002) and by the China Postdoctoral Science Foundation (No. 20110490208).

Numerous hardware data prefetchers have been developed, including sequential [3], stride [4][7][8], Markov [9] and delta correlation (DC) [10][11] prefetchers. Stride prefetchers [7][8] utilize a small reference prediction table (RPT) to record a last miss address and a stride for each stride pattern, and maintain status bits in each RPT entry for obtaining the last prefetched address to avoid redundant prefetches [12]. Stride prefetchers occupy little storage, but cannot handle irregular access patterns. DC prefetchers [10][11] can handle both regular and complicated patterns, and show advantages in performance [13]. DC prefetchers use a global history buffer (GHB) to hold recent miss addresses in FIFO order and link addresses of the same local stream into lists. An index table (IT) is used to identify local streams and point to the address lists in GHB. Multiple GHB entries may be occupied to capture a stride pattern, which can be represented by two consecutive miss addresses, thus wasting the storage. Moreover, DC prefetchers cannot identify whether an address has been prefetched, thus issuing a large number of redundant prefetches.

In this paper, we propose the stride/DC (S/DC) prefetcher which combines and refines ideas of stride and DC prefetchers to achieve better performance and energy efficiency. S/DC uses a pattern prediction table (PPT) to replace the IT. To improve the storage efficiency, PPT records two recent miss addresses in each entry to represent the stride pattern, which prevents a stride pattern from occupying multiple GHB entries. Meanwhile, each PPT entry maintains a 2-bit counter for stride patterns to obtain the last prefetched address in order to avoid redundant prefetches. When handling other patterns, S/DC compares the new predicted address with earlier generated addresses in the prefetch queue to filter the redundant ones. Furthermore, a bit-vector prefetch filter which approximately records addresses evicted from the prefetch queue is used in S/DC to expand the filtering scope.

This paper makes the following contributions. First, S/DC uses the PPT instead of the GHB to capture stride patterns. PPT avoids recording more than two miss addresses for a stride pattern to improve the storage efficiency. Compared with the CZone/DC (C/DC) prefetcher [11], S/DC achieves comparable performance with only 24% of the storage. Second, S/DC filters redundant prefetches by recording the last prefetched address for stride patterns and comparing the new predicted address with earlier generated addresses in the prefetch queue and prefetch filter for other patterns. Thus, it reduces the bandwidth requirements and energy consumption of prefetching. Compared with C/DC, S/DC decreases the percentage of redundant prefetches from 71.04% to 13.01%, and thus reduces 11.46% of the L2 cache energy.

II. Related Work

Many prefetching schemes have been proposed in the past. The simplest one is sequential prefetching [3] which issues a prefetch for the next block when a cache miss happens. Stride prefetchers [7][8] capture sequences of addresses that differ by a constant stride, and prefetch addresses that continue the stride pattern. *K* cache blocks are prefetched when a stride pattern is first captured, and then an additional cache block is prefetched when a previously prefetched block is consumed by the processor [12], where *K* is the prefetch degree.

Correlation prefetching schemes detect correlations among miss addresses to handle complicated access patterns. Markov prefetcher [9] captures repetitive subsequences in the global miss address stream to predict irregular prefetch addresses. It requires large correlation tables to be effective, which makes it hard to implement [10]. The difference between consecutive addresses in the miss address stream comprises the delta stream. Delta correlation (DC) prefetchers [10][11] capture repetitive subsequences in the delta stream to generate prefetch addresses. DC prefetchers use the most recent delta pair (two consecutive deltas) as the correlation key to search the delta stream in reverse order for the same delta pair. If a match is found, prefetch addresses can be computed by the current miss address and the deltas following the match. A recent study shows that DC prefetchers are the top performers among the ten prefetching schemes [13].

To improve the predictability, it is common to divide the global miss address stream into multiple local streams and capture repetitive access patterns in each local stream [14]. The global stream can be split by the program counter (PC) of memory instructions [7][10] or concentration zones (CZones) [8][11]. The physical memory is divided into fixed size ranges called CZones according to the higher order bits of miss addresses [11]. CZones are suitable for prefetchers in low levels of the memory hierarchy since PCs are generally invisible to them. Since out-of-order processors can tolerate most of the short-latency misses [10][11], our proposed S/DC is designed to prefetch into the last level cache (in our case the L2) and uses CZones to divide the global stream.

Redundant prefetches waste the bandwidth and energy, and even incur performance loss due to the resource competition. A recent work [6] uses miss status holding registers (MSHRs) and a Bloom filter to filter redundant prefetches. In this paper, we implement multiple filtering mechanisms based on the structure of S/DC to eliminate redundant prefetches.

III. Motivation

To optimize a processor for energy efficiency, the design choice of each component should be reconsidered. First, low power design is recommended to save energy while achieving comparable performance. Thus, wasted storage and redundant operations should be eliminated. Second, existing resources should be used effectively to improve performance without increasing energy. DC prefetchers [10][11] show advantages in performance, but perform poorly in energy efficiency due to wasted storage and redundant prefetches, which motivates us to propose a novel energy efficient prefetcher.

DC prefetchers contain an IT and a GHB. Each IT entry represents a local stream and contains the *Tag* of the stream and the *Ptr* that points into the list of miss addresses belonging to this stream in GHB. GHB stores recent miss addresses in FIFO manner, and links entries that belong to the same IT entry by pointers to capture repetitive access patterns.

Generally, GHB stores all the miss addresses belonging to an IT entry under the limitation of capacity. Therefore, several GHB entries may be occupied to capture a stride pattern which can be represented by two consecutive miss addresses, thus wasting the storage. We evaluate the occupancy of GHB by stride patterns in the C/DC prefetcher [11], as show in Fig. 1. The results show that on average 53.28% of GHB entries are occupied by stride patterns. If each IT entry is extend with two recent miss address fields to represent a stride pattern, GHB entries wasted by stride patterns can be saved. Thus, GHB are only used for other patterns and its size can be reduced to save energy and area. When handling other patterns, these two fields can work together with GHB. They record most recent miss addresses, and shift earlier miss addresses into GHB.

The reason why DC prefetchers issue a large number of redundant prefetches is that they cannot identify whether an address has been prefetched [6]. Therefore, the key to reduce redundant prefetches is recording earlier issued addresses and comparing the new predicted addresses with them to eliminate the redundant ones. For stride patterns, a counter can be stored in each IT entry to compute the last prefetched address, thus avoiding redundant prefetches. For other patterns, the new predicted addresses can be compared with earlier generated addresses in the prefetch queue to filter the redundant ones. To expand the filtering scope, additional structure can be used to record addresses evicted from the prefetch queue.

IV. Stride/Delta Correlation (S/DC) Prefetcher

In this section, we propose the design and implementation of S/DC. To improve the storage efficiency, S/DC uses a PPT which records two consecutive miss addresses in each entry to capture stride patterns, thus preventing a stride pattern from

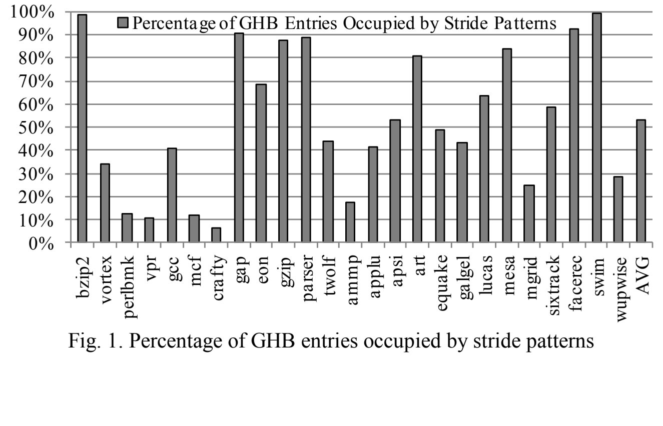

Fig. 1. Percentage of GHB entries occupied by stride patterns

occupying multiple GHB entries. Meanwhile, PPT reduces redundant prefetches when handling stride patterns by storing a counter in each entry to compute the last prefetched address. When handling other patterns, S/DC uses the prefetch queue and prefetch filter to store earlier generated prefetch addresses together in order to eliminate redundant prefetches.

A. Architecture

S/DC is designed to prefetch into the L2 cache, and divides miss addresses into local streams according to CZones. The structure of the proposed S/DC is shown in Fig. 2. Next, we introduce each of the key components in detail.

PPT utilizes the fully or set associative structure and each entry contains six fields, as shown in Fig. 2. The *Tag* is used to identify different CZones. Two most recent miss indexes of the CZone are stored in the *LastIdx1* and *LastIdx0*, which are enough to represent stride patterns. When handling other DC patterns, earlier miss addresses are shifted into GHB and the *Ptr* points to the head of the address list in GHB. The *State* of the entry may be *INV*, *INIT*, *STRIDE* and *DELTA*, which will be introduced in section IV.B. The *Cnt* is a 2-bit saturation counter which records the match times of the stride pattern.

GHB is a FIFO-like circular buffer that holds recent miss addresses shifted from PPT in chronological order, as shown in Fig. 2. Since the tag bits of miss addresses can be found in PPT, GHB only needs to store the index bits in the *MissIdx* field. The *Ptr* field stores pointers which chain the GHB entries that belong to the same CZone into address lists. The *LastIdx1* and *LastIdx0* in the PPT entry and the address list in GHB comprise the miss address stream. The delta stream is computed in sequential manner when searching the address stream, and stored in the delta buffer in reverse order to generate the prefetch addresses when a DC pattern is captured.

Prefetch queue is a 32-entry FIFO-like circular buffer that holds recent generated prefetch addresses and issues them to the L2 cache in sequential manner, as shown in Fig. 2. Each entry contains three fields. The *Addr* field stores the prefetch address. The *V* and *Issued* fields are the status of the address. Only valid and unissued addresses can be sent to the L2 cache. Issued addresses are only used to filter redundant prefetches.

Prefetch filter is a bit-vector that is indexed with the output of the exclusive-or operation of the lower and higher order bits of the cache block address, as shown in Fig. 2. When an issued prefetch address is replaced from the prefetch queue, the filter is accessed with that address and the corresponding bit is set. Prefetch filter is used as a supplement to the prefetch queue to eliminate redundant prefetches.

To keep prefetched blocks from modifying the original L2 demand miss address stream, a prefetch bit is added to each L2 cache block [3][11]. This bit is set when a prefetched block is written into the L2 cache. When an access hits a cache block with a set prefetch bit, the prefetch bit is cleared and the hit address is used to access the S/DC as if it was a L2 miss.

B. Main Flow of Prefetching

When a L2 miss occurs (or an access hits a prefetched but not yet accessed cache block), the tag bits of the miss address are used to search the PPT. Fig. 3 shows the state machine of the PPT entry. If the tag fails to match any of the valid tags currently held in the PPT, the miss is not considered to belong to any of the known CZones and an *INV* entry or the least recently used (LRU) entry is allocated in the PPT. The *Tag* and *LastIdx1* fields are updated with the tag and index of the miss address, and then the state of the entry changes to *INIT*. Subsequent misses with the same tag are considered to belong to the same CZone and cause the update of the entry.

In *INIT* state, when a subsequent miss occurs, the *LastIdx1* is shifted into the *LastIdx0* and updated with the index of the new miss address. And then the *Cnt* is cleared. Afterwards, the entry enters *STRIDE* state.

In *STRIDE* state, when a subsequent miss occurs, the last and new strides are computed by the *LastIdx0*, the *LastIdx1* and the new miss index. If the new stride is identical with the last stride, a stride pattern is captured and the *Cnt* is increased. If the *Cnt* is 1, it means the stride pattern is first captured and a number of prefetches determined by the prefetch degree are issued; otherwise a prefetch for the next unprefetched cache block is issued. The *LastIdx1* is shifted into the *LastIdx0* and updated with the new miss index. The entry stays at *STRIDE* state and no address is inserted into the GHB. If the new stride differs from the last stride, the access pattern is not a stride

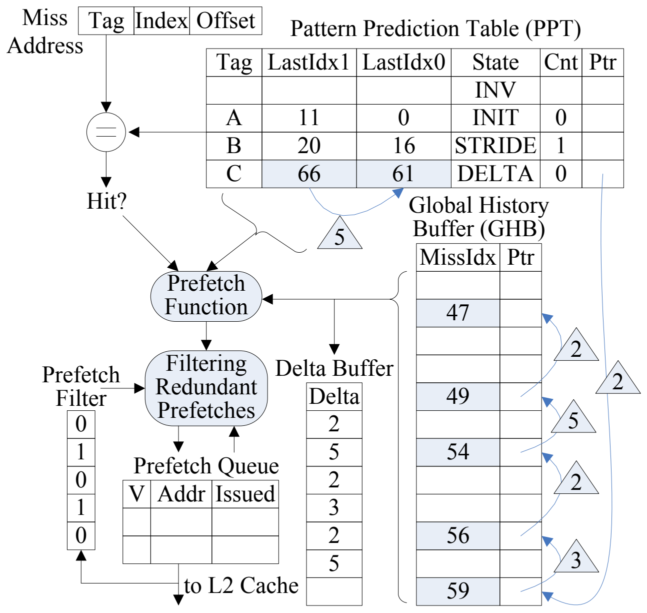

Fig. 2. The structure of the proposed S/DC prefetcher

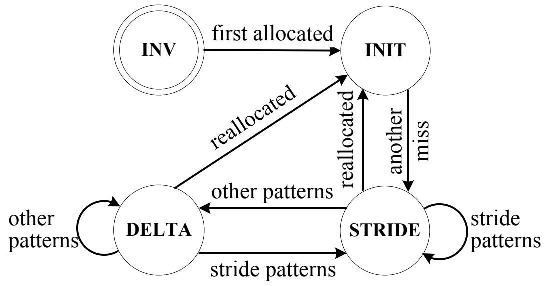

Fig. 3. The state machine of the PPT entry

pattern. If the *Cnt* is not 0, *Cnt* discarded addresses computed by the *LastIdx0* and the last stride are inserted into the GHB to restore the original miss stream. And then the *LastIdx0* is shifted into the GHB and the new miss index is shifted into the *LastIdx1*. The *ptr* is also updated. Afterwards, the entry enters *DELTA* state to detect other DC patterns.

In *DELTA* state, when a subsequent miss occurs, the key delta pair is computed by the *LastIdx0*, the *LastIdx1* and the new miss index. If the two deltas of the delta pair are the same, then a stride pattern is captured. The *LastIdx1* and *LastIdx0* are updated, while the *Cnt* is set to 1. And then the entry enters *STRIDE* state to issue prefetches. Otherwise, the entry stays at *DELTA* state. The *LastIdx0* is shifted into the GHB and the new miss index is shifted into the *LastIdx1*. S/DC uses the key delta pair to search the miss address stream in reverse order. If a match is found, prefetch addresses are computed by the current miss address and the deltas stored in the delta buffer.

C. Redundant Prefetch Filtering Mechanisms

S/DC presents three filtering mechanisms for different access patterns. For stride patterns, S/DC uses the *Cnt* in each PPT entry to compute the last prefetched address, and thus only issues prefetches for unprefetched cache blocks to avoid redundant prefetches. For other patterns, S/DC uses the new predicted address to search the prefetch queue and prefetch filter simultaneously before issuing a prefetch. If a match is found in either of the two structures, the prefetch is redundant and discarded. To limit false-positive matches caused by the exclusive-or operation, the prefetch filter is reset periodically (every 100 accesses to the prefetch filter in this paper).

V. EXPERIMENTS

In this section, we evaluate the performance and energy of S/DC using the SimpleScalar Alpha [15] simulator which is extended with the Wattch [16] power model. We use SPEC CPU2000 benchmarks [17] for evaluation. For each program, a representative sample of 100 million instructions selected by the SimPoint [18] is run with reference inputs. Table 1 summarizes the configuration of the baseline processor model which is a typical 4-issue superscalar processor. We propose three versions (S/DC_conf0, S/DC_conf1 and S/DC_conf2) of S/DC to separately evaluate the effectiveness of the PPT, the prefetch queue and the prefetch filter. The CZone/stride prefetcher, namely stream prefetcher (SP) [4][8], and the CZone/DC (C/DC) [11] prefetcher are used for comparison. The parameters of the above prefetchers are presented in Table 2. The prefetch degree is set to 4 for all the prefetchers.

A. Storage Efficiency and Performance

S/DC uses the PPT to capture stride patterns in order to avoid recording more than two miss addresses for a stride pattern, thus reducing the storage without lowering the performance. To quantify the effectiveness, we choose five GHB options (32-, 64-, 128-, 256- and 512-entry) to evaluate the performance improvement of C/DC and S/DC with respect to the baseline processor, as shown in Fig. 4 and Fig. 5. The

Table 1. The configuration of the baseline processor

Feature	Parameters
Pipeline	8 stages, 4-issue/decode/commit, 1GHz
Instruction Window	128-entry
Load/Store Queue	64-entry
L1 I/DCache	64KB, 4-way, 32-byte line, 2 cycles
L2 Cache	2MB, 16-way, 32-byte line, 12 cycles
Main Memory	200-cycle latency

Table 2. The configuration of prefetchers

Method	Parameters
SP	32-entry reference prediction table (RPT)
C/DC	32-entry index table (IT), 512-entry GHB
S/DC_conf0	32-entry PPT, 64-entry GHB, no other filtering mechanism
S/DC_conf1	32-entry PPT, 64-entry GHB, filtering through the prefetch queue
S/DC_conf2	32-entry PPT, 64-entry GHB, filtering through the prefetch queue and prefetch filter

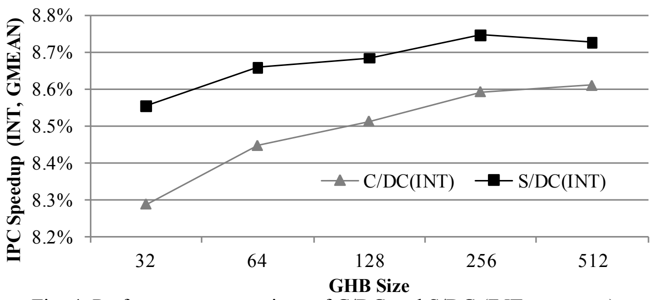

Fig. 4. Performance comparison of C/DC and S/DC (INT programs)

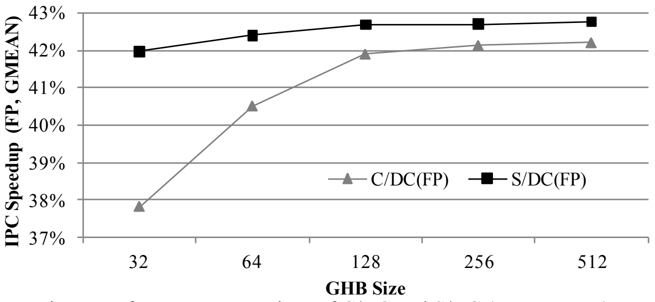

Fig. 5. Performance comparison of C/DC and S/DC (FP programs)

results indicate that both the performance of C/DC and S/DC are improved as the GHB size increases, and S/DC always outperforms C/DC with the same GHB size. As the GHB size decreases, the advantage of S/DC is growing, since S/DC can prevent stride patterns from occupying GHB entries. As for S/DC, performance is slightly lowered from 256- to 512-entry GHB. This is because 512-entry GHB holds more stale data than 256-entry GHB and stale data incurs useless prefetches and cache pollution in a few programs.

In particular, S/DC(64-entry GHB) achieves comparable performance with C/DC(512-entry GHB), as shown in Fig. 4 and Fig. 5. We divide the 32-bit address into 16-bit *Tag*, 11-bit *Index* and 5-bit *Offset* and compare the storage of C/DC

Table 3. Storage comparison of C/DC and S/DC

Prefetcher		Organization	Capacity
C/DC (512)	IT	(16-bit *Tag* + 9-bit *Ptr*)×32	~1.35KB
	GHB	(11-bit *MissIdx* + 9-bit *Ptr*)×512	
S/DC (64)	PPT	(16-bit *Tag* + 11-bit *LastIdx1* + 11-bit *LastIdx0* + 2-bit *State* + 2-bit *Cnt* + 6-bit *Ptr*)×32	~0.32KB
	GHB	(11-bit *MissIdx* + 6-bit *Ptr*)×64	

and S/DC in this situation, as shown in Table 3. The results indicate that S/DC achieves comparable performance with about 24% of the storage as compared to C/DC.

We show the performance improvement of the different prefetchers over the baseline processor in Fig. 6 and Fig. 7. On average, the performance speedup of SP, C/DC(512-entry GHB), S/DC(64-entry GHB) and S/DC(256-entry GHB) is 7.64%, 8.61%, 8.64% and 8.75% for INT programs, and 21.54%, 42.21%, 42.23% and 42.70% for FP programs. The results demonstrate that C/DC and S/DC can further improve the performance in all programs except *mcf* by handling both regular and complicated access patterns as compared to SP. For *mcf*, 2MB L2 cache fits the working set and repetitive access patterns mostly hit the L2 cache. Therefore, the prefetch accuracy of C/DC and S/DC is low, which causes a great deal of cache pollution and lowers the performance. When we evaluate *mcf* with 512KB L2 cache, the results show that C/DC and S/DC can improve the performance of the baseline processor observably (about double the performance). Compared with C/DC(512-entry GHB), S/DC(64-entry GHB) evaluated in the following parts of this paper achieves comparable or better performance with only 24% of the storage in most of the programs, and S/DC(256-entry GHB) achieves comparable or better performance in all programs.

B. Redundant Prefetches and Energy

S/DC implements three filtering mechanisms through the PPT, prefetch queue and prefetch filter to eliminate redundant prefetches, thus reducing the bandwidth requirements and energy consumption of prefetching. Fig. 8 and Fig. 9 show the prefetch times of the five prefetchers, including new and redundant prefetches, normalized to the prefetch times of C/DC. On average, the percentage of redundant prefetches of SP, C/DC, S/DC_conf0, S/DC_conf1 and S/DC_conf2 is 13.06%, 70.35%, 34.68%, 17.23% and 17.07% for INT programs, and 9.19%, 71.69%, 41.40%, 10.34% and 9.26% for FP programs. The results indicate that both the C/DC and S/DC issue more prefetches and redundant prefetches than SP to achieve higher performance. Compared with C/DC, S/DC significantly reduces the redundant prefetches through the three filtering mechanisms while issuing comparable new prefetches. In particular, S/DC_conf0 only avoids generating redundant prefetches when handling stride patterns, and S/DC_conf1 can further eliminate most of the redundant prefetches generated when handling other DC patterns. In S/DC_conf2, the prefetch filter is added as a supplement to the prefetch queue to further discard redundant prefetches. The results demonstrate that the prefetch filter is less effective than the prefetch queue, because prefetches generally overlap with recent rather than earlier issued prefetches. On the other hand, false-positive matches of the prefetch filter may lower the performance, so it is an optional mechanism in S/DC.

When a prefetch is issued by the prefetcher, the L2 cache is

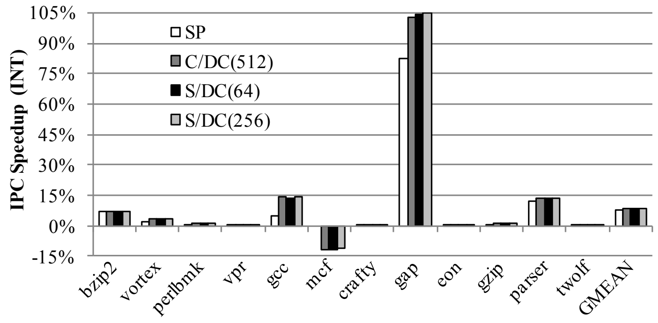

Fig. 6. Performance speedup of the four prefetchers (INT programs)

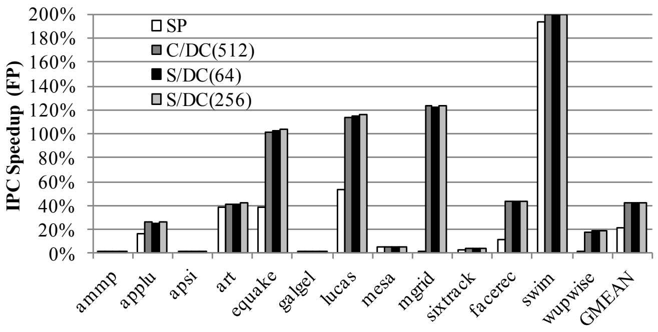

Fig. 7. Performance speedup of the four prefetchers (FP programs)

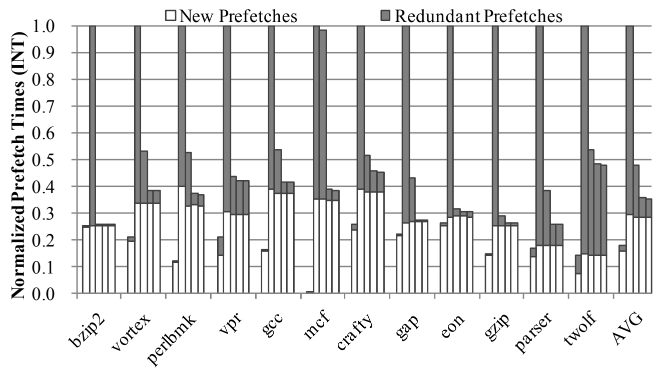

Fig. 8. Redundant prefetches of the five prefetchers (from left to right: SP, C/DC, S/DC_conf0, S/DC_conf1 and S/DC_conf2, INT programs)

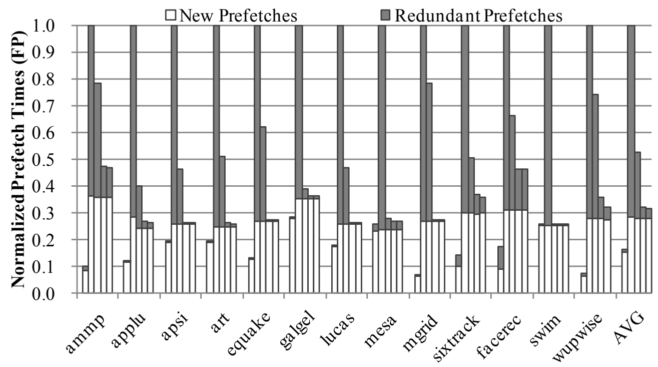

Fig. 9. Redundant prefetches of the five prefetchers (from left to right: SP, C/DC, S/DC_conf0, S/DC_conf1 and S/DC_conf2, FP programs)

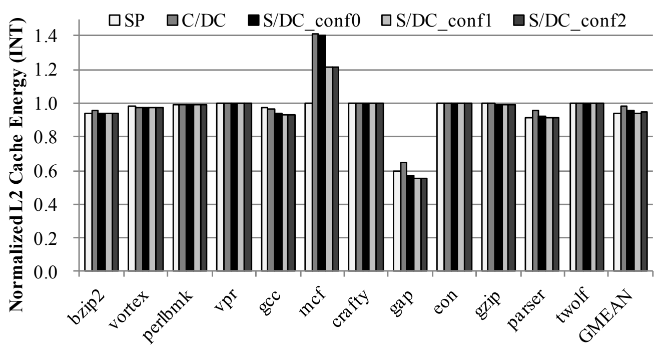

Fig. 10. L2 cache energy of the five prefetchers (INT programs)

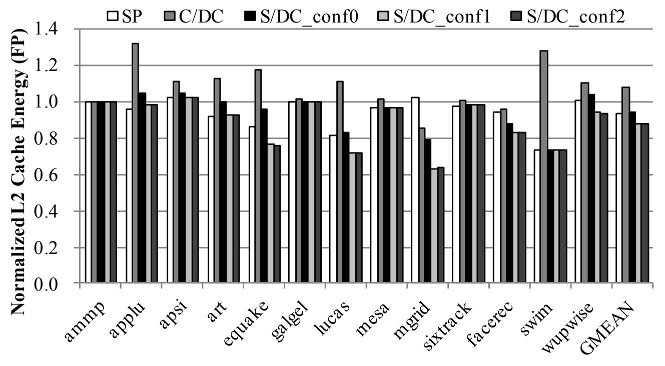

Fig. 11. L2 cache energy of the five prefetchers (FP programs)

accessed first to check if the prefetch address is already in the cache. From Fig. 8 and Fig. 9, we can see that S/DC reduces the overall number of prefetches by discarding a large number of redundant prefetches, indicating that the number of L2 cache accesses issued by the prefetcher is also decreased, so S/DC can reduce the energy of the L2 cache. To quantify the effectiveness, we evaluate the L2 cache energy (including dynamic and leakage energy) of the five prefetchers, normalized to the L2 cache energy of the baseline processor, as shown in Fig. 10 and Fig. 11. On average, SP, C/DC, S/DC_conf0, S/DC_conf1 and S/DC_conf2 reduces the L2 cache energy by 5.89%, 1.99%, 3.97%, 5.57% and 5.55% for INT programs, and 6.48%, -7.40%, 6.23%, 12.57% and 12.64% for FP programs. Compared with C/DC, S/DC_conf2 reduces the L2 cache energy by an average of 11.46% for all programs. The results demonstrate that S/DC can significantly reduce the bandwidth requirements and energy consumption of prefetching by filtering redundant prefetches.

VI. Conclusion

In this paper, we propose a storage and energy efficient data prefetcher called S/DC. To improve the storage efficiency, S/DC presents the PPT which stores two consecutive miss addresses in each entry to represent the stride pattern, and thus avoids recording more than two miss addresses for a stride pattern. Meanwhile, PPT stores a counter in each entry to compute the last prefetched address for stride patterns to avoid redundant prefetches. To reduce redundant prefetches when prefetching other patterns, S/DC compares the new predicted address with earlier generated addresses in the prefetch queue and filters the redundant ones. Furthermore, to expand the filtering scope, S/DC uses a bit-vector prefetch filter which approximately records addresses evicted from the prefetch queue to further reduce redundant prefetches.

Our experimental results demonstrate that S/DC achieves comparable performance with only 24% of the storage as compared to the C/DC prefetcher. Furthermore, compared with C/DC, S/DC can reduce the percentage of redundant prefetches and the number of L2 cache accesses significantly, thus reducing the bandwidth requirements for accessing the L2 cache and decreasing the L2 cache energy.

References

[1] S. Borkar, and A. A. Chien, "The future of microprocessors", *Communications of the ACM*, vol. 54, no. 5, pp. 67-77, May 2011.

[2] O. Azizi, A. Mahesri, et al., "Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pp. 26-36, June 2010.

[3] S. P. Vanderwiel, and D. J. Lilja, “Data prefetch mechanisms,” *ACM Computing Surveys*, vol. 32, no. 2, pp. 174-199, June 2000.

[4] H. Q. Le, W. J. Starke, et al., “IBM POWER6 microarchitecture,” *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 639–662, November 2007.

[5] Y. Guo, P. Narayanan, et al., “Energy-efficient hardware data prefetching,” *IEEE Transactions on Very Large Scale Integration Systems*, vol. 19, no. 2, pp. 250-263, February 2011

[6] M. Dimitrov, and H. Zhou, “Combining local and global history for high performance data prefetching,” *Journal of Instruction-Level Parallelism*, vol.13, pp. 1-14, 2011.

[7] T. F. Chen, and J. L. Baer, “Effective hardware-based data prefetching for high-performance processors,” *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609-623, May 1995.

[8] S. Palacharla, and R. E. Kessler, “Evaluating stream buffers as a secondary cache replacement,” in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp. 24-33, 1994.

[9] D. Joseph, and D. Grunwald, “Prefetching using Markov predictors,” in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 252-263, June 1997.

[10] K. J. Nesbit, and J. E. Smith, “Data cache prefetching using a global history buffer,” in *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, pp. 90-97, 2004.

[11] K. Nesbit, A. Dhodapkar, and J. E. Smith, “AC/DC: An adaptive data cache prefetcher,” in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 135-145, September-October 2004.

[12] S. Iacobovici, L. Spracklen, et al., “Effective stream-based and execution-based data prefetching,” in *Proceedings of the 18th Annual International Conference on Supercomputing*, pp. 1-11, 2004.

[13] D. G. Perez, G. Mouchard, and O. Temam, “MicroLib: A case for the quantitative comparison of micro-architecture mechanisms,” in *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pp. 43-54, December 2004.

[14] P. Díaz and M. Cintra, “Stream chaining: Exploiting multiple levels of correlation in data prefetching,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pp. 81-92, 2009.

[15] T. Austin, E. Larson, and D. Ernst, “SimpleScalar: An infrastructure for computer system modeling,” *IEEE Computer*, vol. 35, no. 2, pp. 59-67, February 2002.

[16] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: A framework for architectural-level power analysis and optimizations,” in *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 83-94, June 2000.

[17] J. L. Henning, “SPEC CPU2000: Measuring CPU performance in the new millennium,” *IEEE Computer*, vol. 33, no. 7, pp. 28-35, July 2000.

[18] E. Perelman, G. Hamerly, and B. Calder, “Picking statistically valid and early simulation points,” in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pp. 244–255, September 2003.