2013 IEEE Eighth International Conference on Networking, Architecture and Storage

Spontaneous Reload Cache: Mimicking a Larger Cache with Minimal Hardware Requirement

Lunkai Zhang[1,2], Mingzhe Zhang[1,3], Lingjun Fan[1,2], Da Wang[1], Paolo Ienne[4]

[1]*State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences*
[2]*University of Chinese Academy of Sciences*
[3]*College of Computer Science, Inner Mongolia University*
[4]*École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland*

{zhanglunkai, zhangmingzhe, fanlingjun, wangda}@ict.ac.cn, Paolo.Ienne@epfl.ch

***Abstract*—In modern processor systems, on-chip Last Level Caches (LLCs) are used to bridge the speed gap between CPUs and off-chip memory. In recent years, the LRU policy effectiveness in low level caches has been questioned. A significant amount of recent work has explored the design space of replacement policies for CPUs' low level cache systems, and proposed a variety of replacement policies. All these pieces of work are based on the traditional idea of a conventional passive cache, which triggers memory accesses exclusively when there is a cache miss. Such passive cache systems have a theoretical performance upper bound, which is represented by Optimal Algorithm. In this work, we introduce a novel cache system called Spontaneous Reload Cache (SR-Cache). Contrary to passive caches, no matter whether a cache access is a hit or miss, an SR-Cache can actively load or reload an off-chip data block which is predicted to be used in the near future and evict the data block which has the lowest probability to be reused soon. We show that, with minimal hardware overhead, SR-Cache can achieve much better performance than conventional passive caches.**

***Keywords*-processor cache; replacement policy; spontaneous reload;**

I. Introduction

The imbalance of the speed between processors and off-chip memory (also know as *memory wall*) is one of the most severe challenge in modern processor industry. To fight the memory wall, modern processors often adopt relatively large *Last Level Caches* (LLCs), hoping that the needed data can be captured by LLCs without accessing the off-chip memory. However, some applications have working sets too large to fit LLCs, thus LLCs get quite easily thrashing and fail to defend the processors against the memory wall.

In this work, we try to alleviate the LLC thrashing problem by introducing the *Spontaneous Reload Cache* (SR-Cache). Contrary to a conventional cache which is a passive module which can only trigger off-chip memory access under cache misses, an SR-Cache is an active module which can spontaneously reload previously accessed data from off-chip memory before new accesses. The SR-Cache manages to mimic a larger cache with much smaller hardware overhead.

The rest of paper is organized as follows: Section II elaborates on the motivation of SR-Cache. Section III gives the architecture of a static SR-Cache. Section IV discusses a problem of the static SR-Cache, and proposes a dynamic SR-Cache to solve this issue. Section V presents the experimental methodology and Section VI shows the results of our experiments. Section VII discusses related prior work and in Section VIII we conclude our work.

II. Motivation

Acting as a passive module which can trigger memory accesses only when encountering access misses, the conventional cache has a theoretical performance upper bound which can be achieved by Belady's Optimal (OPT) policy [1]. In Figures 1– 4, we compare the different cache architecture's effectiveness with a simplified cyclic access pattern $(A1, A2, A3, A4)^N$ to one cache set. We analyze the caches' behavior in terms of *Reuse Distance* (first introduced by Petoumenos et al.'s work [3]), which indicates how many accesses to the cache set will take place before the cache block is accessed again.

The OPT policy can guarantee 50 percent hit rate, as shown in Figures 1. Because the OPT policy is theoretically the best policy among all cache replacement policies, we have to break some "rules" of the conventional cache in order to get better performance than the conventional cache with OPT policy.

At the same time, a 4-way cache, no matter what policy it adopts, will actually get no capacity miss when dealing with the same access pattern. What we are interested in is the behavior of the cache blocks' reuse distance: We can tell from Figure 2 that only the cache block with the smallest reuse distance will be accessed the next time (these blocks are marked with red ovals in Figure 2), while other cache blocks are "useless" on the next cache access time.

For the purpose of saving on-chip storage, we might use an active cache module, which can automatically bring in the cache blocks with small reuse distance and evict the cache blocks with larger reuse distance. We call such active cache a *Spontaneous Reload Cache* (SR-Cache). Figure 3 gives a

978-0-7695-5034-3/13 $26.00 © 2013 IEEE
DOI 10.1109/NAS.2013.11

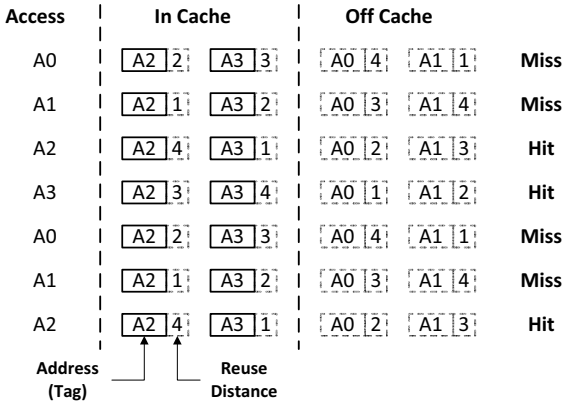

Figure 1: **Behavior of a 2-Way OPT Policy Conventional Cache.** For the given pattern, this cache guarantees a 50% hit rate, which is the theoretical upper bound of conventional caches.

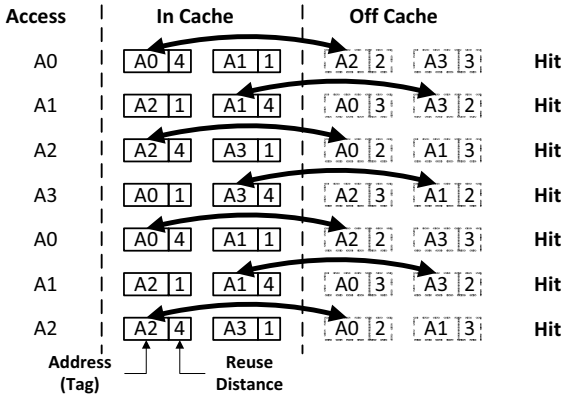

Figure 3: **Behavior of a 2-Way Ideal SR-Cache.** This figure shows an ideal SR-Cache which can spontaneously reload the off-cache block which will be reused soon, and evict the in-cache block which will be reused far away. However, since the reuse distance information will be lost when a block is evicted, additional hardware is needed to store such information.

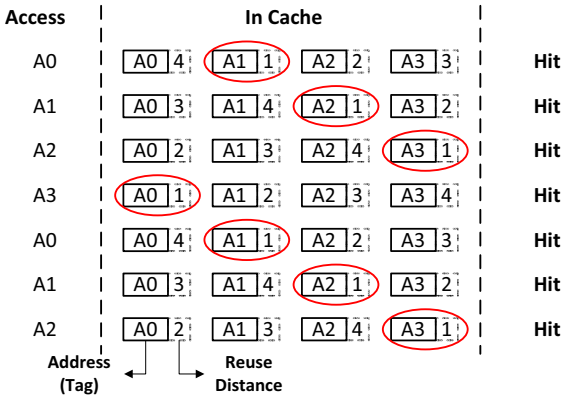

Figure 2: **Behavior of a 4-Way Conventional Cache.** Only the cache block with the smallest reuse distance will be reused in the next cache access, and other cache blocks are somewhat "useless" in the next cache access.

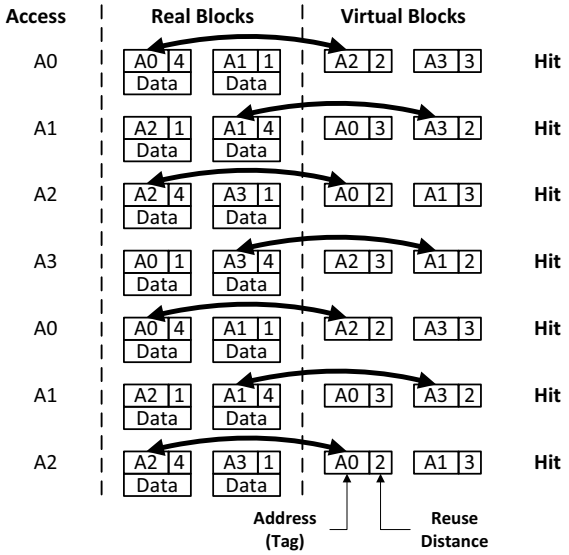

Figure 4: **Behavior of a 2-Way Practical SR-Cache.** To solve the problem in Figure 3, we use virtual cache blocks, which do not contain a data field, to store estimated reuse information of off-chip cache blocks. For the given access pattern, this 2-way SR-Cache can achieve the performance a 4-way conventional cache, with much less storage.

2-way example of SR-Cache using OPT policy. The figure suggests that, using the OPT policy, the 2-way SR-Cache can mimic ideally a 4-way cache.

However, if we try to build a practical SR-Cache, two problems need to be solved:

- The OPT policy needs to look into the future, which is impossible in a real situation. Thus, some substitute of the OPT policy is needed.
- In conventional caches, a cache block's reuse distance information would be lost when the block is evicted from the cache. Thus, we need some additional hardware structure to store this otherwise lost information.

To approximate the OPT policy, we directly predict the reuse time of a cache block. Every cache block in the SR-Cache has a *Reuse History Field* which helps predicting the reuse distance of the block. For the second problem, we use *virtual cache blocks*, which are cache blocks without data fields, to preserve the address and reuse distance information of

currently off-chip data. As shown in Figure 4, every cache set in a practical SR-Cache has both real cache blocks and virtual cache blocks. On every cache reference, the SR-Cache performs reload operation if some virtual block's reuse distance is smaller than any real block's reuse distance.

III. Implementation of Static SR-Cache

A. The Architecture of SR-Cache Set

As shown in Figure 5, a Static SR-Cache Set mainly consists of two parts: real ordinary blocks with data and virtual blocks without data. Both real and virtual blocks contain a Reuse History Field, which records the statistics used to predict the cache blocks' *Reuse Distance*. This *Reuse History Field* consists of two sub-fields: the *Idle Count Field*, which records the cache-set access count since the last access, and the *Reuse Interval Field*, which gives a predicted cache-set access count between the next access in future and last access.

To reduce storage overhead and power consumption, we update the cache blocks' *Idle Count Field* every **few** cache set accesses instead of every cache access. To do this, in every cache-set, we add a cyclic counter which keeps count of the cache-set accesses. These cyclic counters trigger the update operation of corresponding cache blocks' *Reuse History Field* every few accesses of its cache set.

B. Detailed Behavior of SR-Cache

To implement a practical SR-Cache, we should change as little as possible to the conventional cache's architecture and behavior. Figure 7 gives the work flow of a SR-Cache when it is dealing with a cache access. Compared to the conventional cache's work flow shown in Figure 6, the SR-Cache does an additional reload check, irrespectively of the current cache access being a hit or a miss. If the SR-Cache determines that it is necessary, it will issue a reload access to the main memory to get the data back into the cache. In the rest of this section, we will discuss the detailed behavior of the SR-Cache.

1) Predicting an SR-Cache Block's Reuse Distance.: It is of great importance to find a proper function which can accurately predict the Reuse Distance of the blocks. Here we present two *Reuse Distance* functions, and a quantitative comparison of these functions will be shown in Section VI-A.

- **Ideal Reuse Distance Prediction**. If *Idle Count* is smaller than the *Reuse Interval*, *Reuse Distance* is predicted as the difference between *Reuse Distance* and *Idle Count*; and if *Idle Count* is larger than the *Reuse Interval*, *Reuse Distance* is predicted as infinite.
- **Absolute Reuse Distance Prediction.** *Reuse Distance* is predicted as the absolute difference between *Reuse Interval* and *Idle Count*.

2) Looking for the Appropriate Virtual Block to Reload.: SR-Cache selects both the real block with the highest *Reuse Distance* (the real block predicted to be reused in most distant future) and the virtual block with the lowest *Reuse Distance* (the virtual block predicted to be reused in the nearest future). If the *Reuse Distance* of selected real block is larger than the *Reuse Distance* of selected virtual block, the SR-Cache will spontaneously reload the selected virtual block from main memory.

3) Inserting the Currently Accessed Cache Block.: Similar to a conventional cache, the SR-Cache needs to insert the currently accessed cache block to the corresponding cache set. During the insertion operation, the SR-Cache needs to update the block's *Reuse Interval* value. Based on different situations, we may adopt different actions for the currently accessed cache block:

- **Current access hits in a cache block (either real block or virtual block).** In such situation, the SR-Cache predicts the inserted block's *Reuse Interval* to be the *Idle Count* value of the hit block (which is actually the Reuse Interval of the nearest history), and resets the *Idle Count* back to 0.
- **Current access misses in both real and virtual blocks.** In such situation, the SR-Cache will set the current real block's *Idle Count* to 0, but it cannot accurately predict the *Reuse Interval*. In this case, there are two different inaccurate approaches to predict the *Reuse Interval's* value:
  - Predict that the block will be reused soon ($ReuseInterval = 1$).
  - Predict that the block will never be reused ($ReuseInterval = +\infty$).

  The effectiveness of these two prediction will be quantitatively studied in Section VI-A.

IV. Dynamic SR-Cache

The Static SR-Cache described above may encounter two obstacles when dealing with real-world applications:

- The SR-Cache will have considerable more storage overhead than the conventional cache.
- In some applications (e.g., soplex_ref_mps in SPEC2006), the Reuse Distance of the data cannot be accurately predicted (see results in Section VI-A).

Therefore, we introduce a Dynamic SR-Cache. When the current application is not suitable for the SR-Cache, this Dynamic SR-Cache can dynamically configure its virtual blocks back as several real blocks and perform a LRU-like replacement policy. As shown in Figure 8, a Dynamic SR-Cache Set can be configured into two different modes:

- **Spontaneous Reload (SR) mode.** The set behaves exactly as a Static SR-Cache set.
- **Approximate Least Recent Use (ALRU) mode.** The set's virtual block storage is converted to real blocks.

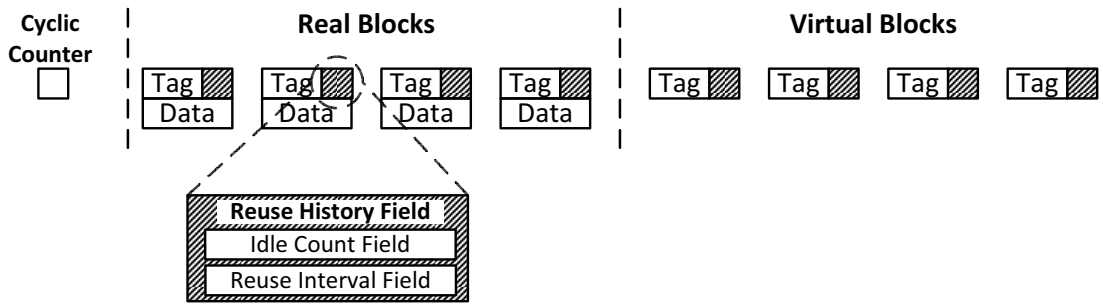

Figure 5: **Architecture of a SR-Cache Set.** One SR-Cache Set mainly consists of two parts: virtual blocks without data field and real blocks; besides, it also has a cyclic counter to record the set access count. Both real and virtual blocks have a *Reuse History Field* to store the information for spontaneous reloading decision. This *Reuse History Field* consists of two sub-fields: *Idle Count Field* and *Reuse Interval Field*.

ALRU mode cache set uses a replacement policy which is an approximation of the LRU policy—when a replacement is needed, the block with largest *Idle Count* is selected as victim.

We use the *Set-Dueling* technique (proposed by Qureshi et al. [2]) to decide the configuration of the Dynamic SR-Cache. That is, a small portion of the Dynamic-SR Cache's sets, called *sampler* sets, are statically configured (half of them configured in SR mode and the other half configured in ALRU mode); the remaining sets, called *follower* sets, are dynamically controlled by a *Policy Selector* (PSEL) to choose between the two modes. PSEL is a saturating counter which keeps track of the hit count difference of the two kinds of sampler sets and its value indicates which policy is better for the current application. Of course, when a *follower* set is changing from ALRU mode to SR mode, to guarantee correctness, the set needs to write back dirty data the real blocks which are about to be converted to virtual blocks.

Changing from SR to ALRU mode causes the cache set to lose all the virtual block information; and changing from ALRU to SR mode causes the cache set to lose several real cache block's data. Thus, frequently changing a cache set's mode incurs considerable overhead. To avoid such ping-pong effect in mode changing, we use a *High Threshold Policy Converting* (HTPC). Instead of immediately switching the cache management policy, the Dynamic SR-Cache with HTPC keeps the current cache management policy, unless the benefit of the opposite policy has crossed this a *ConvertingThreshold* value.

V. Experimental Methodology

We use M5 simulator [12] to simulate an Alpha21264-like processor system. For the cache system, we simulate a two-level cache hierarchy. The L1 I/D-caches are conventional caches using LRU replacement policy. For L2 cache (this is Last Level Cache in our system), we model it as a Dynamic SR-cache system. For comparison, we also simulation L2 cache as a conventional cache with several state-of-the-art replacement policies (LRU, DIP[2], DRRIP[5]).

To compensate the additional storage for the virtual ways, we reduce the number of real ways of the SR-Cache. For a 40-bit address space and 1024-set cache system (that is, the address tag storage is 30 bits), 1 real cache blocks has a similar storage requirement as 14 virtual blocks. Thus, in our experiments, the L2 SR-Cache set is configured to have the same storage requirement as a 16-way conventional cache set (15 real blocks and 14 virtual blocks, 14 real blocks and 28 virtual blocks, etc.). The detailed configuration of the simulator is listed in Table I.

We selected 13 memory intensive benchmarks from the SPEC CPU 2000 & 2006 suites. Table II presents the benchmarks with their *Misses Per 1000 Instructions* (MPKI) of a LRU-policy 1MB ordinary L2 cache.

VI. Experimental Results

A. Impact of Different Spontaneous Reload Schemes

We have proposed two different schemes to calculate a cache block's *Reuse Distance* (in Section III-B1) and two different schemes to initialize a first-time accessed block's *Reuse Interval* (in Section III-B3). We quantitatively compare the effectiveness of four different combinations of these Spontaneous Reload schemes. Here, all the SR-Caches have 14 real ways and 28 virtual ways.

Figure 9 shows the impact of these different schemes on the *Misses Per 1000 Instructions* (MPKI). We can tell from this

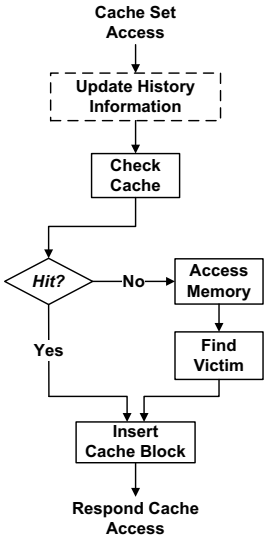

Figure 6: **Work Flow of a Conventional Cache.** At first, the cache determines whether the access hits in the cache or not. If the access is not a hit, the cache should get the data from the off-chip memory. Then, the cache should insert the currently accessed cache block, change the history statistics, and at last respond to the access request with the data.

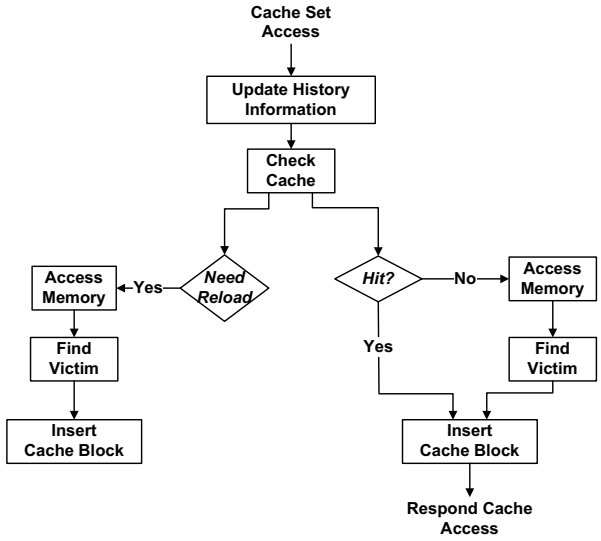

Figure 7: **Work Flow of a SR-Cache.** Besides the same behavior with a conventional cache, SR-Cache should do a possible spontaneous reload in parallel. On every access, no matter whether it is a hit or a miss, the SR-Cache should check the history statistics to decide whether it needs a spontaneous reload or not. If such a reload is needed, the SR-Cache should access the memory and get the data, then insert the returned data into the cache, and at last update the history statistics in the tag storage.

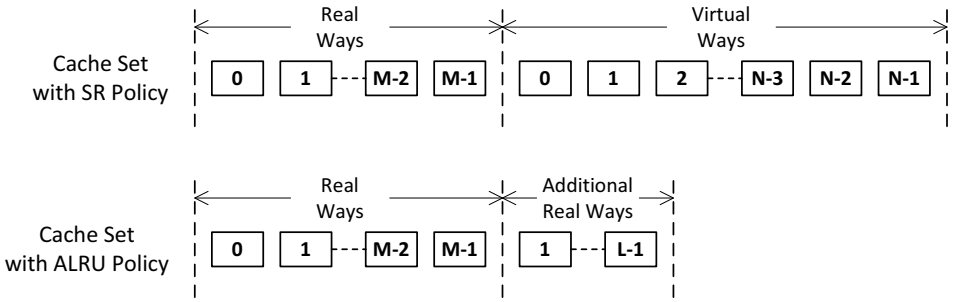

Figure 8: **Two Different Cache Set Modes in Dynamic SR-Cache.** SR mode consists of real ways and virtual ways. And ALRU mode does not have virtual ways, but it reconfigures the virtual ways' storage to form additional real ways.

figure that the best combination is to use *Absolute Reuse Distance* scheme and *Far Reuse Interval Prediction*. *Absolute Reuse Distance* is favored because the *Reuse Distance* is not a very accurate prediction: a block whose Idle Count slightly outnumbers its Reuse Distance is still a possible candidate for reuse in the near future. *Far Reuse Interval Prediction* is favored because SR-Cache mainly targets applications whose working set is larger than the capacity of real blocks, and therefore predicting the first-used block to be reused in the distant future will help to avoid evicting the data which is currently in the real blocks. In the rest of the paper, unless otherwise indicated, the SR-Cache will use such combination as the base configuration.

B. Impact of Different Real-Virtual Ways Combinations

In this subsection we examine the performance of the SR-Cache with different real and virtual ways. Here we use four different combinations, which are [15 real ways, 14 virtual ways], [14 real ways, 28 virtual ways], [13 real ways, 42 virtual ways], [12 real ways, 56 virtual ways]. In all of these 4 combinations, a SR-Cache set uses an amount of storage similar to that of a 16-way conventional cache set. On one hand, having more virtual ways gives the SR-Cache the ability to remember the reuse history of more data blocks; on the other hand, this also reduces the real ways of the SR-Cache, and thus leads to imprecisions. Therefore, a balance between real and virtual way counts is needed.

Core	Alpha ISA, 5-Stage Pipeline, 8-Wide Dispatch/Retirement 256/256 Int/Fp Registers, 48/64-Entry Inst/Data TLBs 6-Int ALU, 2-Int Mul/Div, 4-Fp ALU, 2-Fp Mul/Div 64-Entry IFQ, 64-Entry LSQ, 192-Entry ROB, 40-Bit Physical Address	
L1 I/D Cache	2-Way, 32KB, 64B/Line, 1-Cycle Lat 10 MSHRs, 8-Entry Write Buffer, LRU Replacement Policy	
L2 Cache (LLC)	Conventional Cache	16-Way, 1024-Set, 64B/Line, (Totally 1MB Capacity) 10-Cycle Lat, 20 MSHRs, 8-Entry Write Buffer
	Dynamic SR-Cache	(16-x)-Real Way, (14x)-Virtual Way, $1 \leqslant x \leqslant 4$, 1024-Set, 64B/Line, 10-Cycle Lat, 20 MSHRs, 8-Entry Write Buffer PSEL Saturating Value: +/-1024 PSEL Threshold: 512
Memory	250-Cycle Lat	

Table I: Simulated Hardware Description

401.bzip2_input_source	0.56
429.mcf	26.60
436.cactusADM	3.61
450.soplex_ref_mps	0.72
456.hmmer_retro_hmm	0.27
459.GemsFDTD	7.79
470.lbm	8.51
473.astar_rivers_cfg	0.37
482.sphinx3	4.67
177.mesa	0.60
179.art	17.15
187.facerec	1.44

Table II: MPKI of Benchmarks

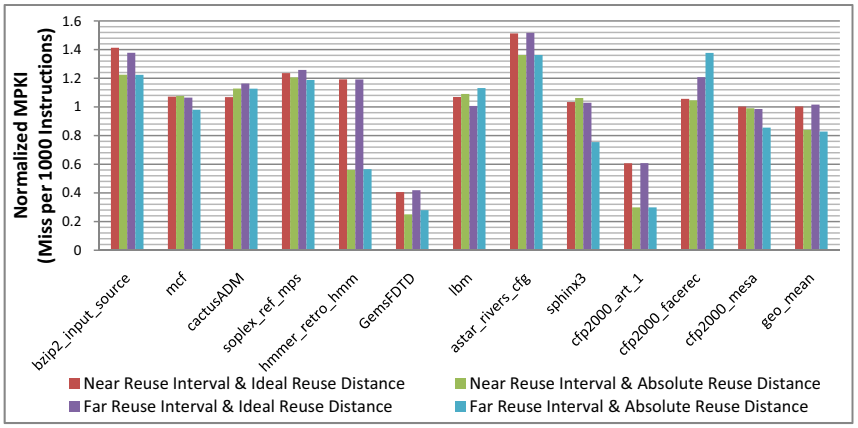

Figure 9: **The impact of different prediction methods of SR-Cache's Reuse Distance and Reuse Interval.** We can see that for most of the benchmarks, Absolute Reuse Distance and Far Reuse Interval are the best combination. Yet, such combination will have considerable inferior performance for some benchmarks. This problem will be solved using a Dynamic SR-Cache.

Figure 10 gives the MPKI of SR-Caches with these four different combinations. We can see that three benchmarks (cfp2000_art_1, hmmer_retro_hmm, cfp2000_mesa) favor fewer virtual ways because these benchmarks' working sets are slightly larger than 1MB and too much more virtual ways have no use. However, 2 other benchmarks (GemsFDTD, sphinx3) use effectively more virtual ways, because they have larger working sets. In the rest of the paper, we select the combination with [14 real ways, 28 virtual ways].

C. Effectiveness of the Dynamic SR-Cache

As we can also tell from the Figure 9, although effective for some of the benchmarks, the Static SR-Cache does perform worse than LRU policy in some benchmarks (bzip2_input_source, cactusADM, soplex_ref_mps, lbm, astar_rivers_cfg, cfp2000_facerec). Thus, in order to avoid such drawback of the Static SR-Cache, we adopt the Dynamic SR-Cache idea (described in Section IV). Figure 11 shows the effectiveness of Dynamic SR-Cache using different techniques.

We first use *Set-Dueling* technique and compare the difference between the Dynamic SR-Caches with and without HTPC (proposed in Section IV). We can tell from Figure 11 that, Dynamic SR-Cache without HTPC fails to reduce the performance lost in benchmarks bzip2_input_source, cactusADM, soplex_ref_mps, lbm and cfp2000_facerec. For Dynamic SR-Cache with HTPC, although successfully reducing the performance lost in some benchmarks, it still fails to make any remarkable improvement for benchmarks cactusADM, soplex_ref_mps; and in benchmark hmmer_retro_hmm, the performance of Dynamic SR-Cache with HTPC considerably falls behind the performance of the Static SR-Cache.

We find that the reason why the Dynamic SR-Cache fails in these benchmarks lies in the inaccuracy of *Set-Dueling* technique. The effectiveness of *Set-Dueling* is based on a premise that all the sets are facing similar cache access behavior. However, in our experiment environment, some benchmarks have remarkable different access behaviors among different cache sets. For example, in a Dynamic SR-

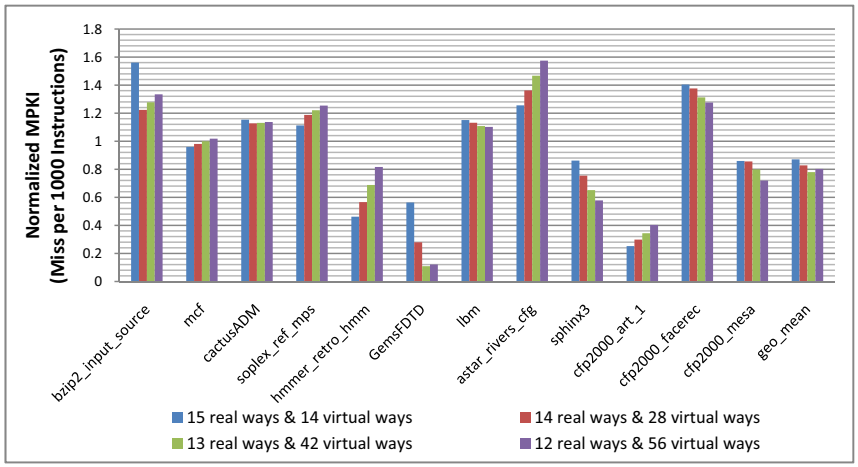

Figure 10: **Impact of Different Real-Virtual Ways Combinations.** This figure gives the MPKI comparison of the SR-Caches with different combinations of real and virtual ways.

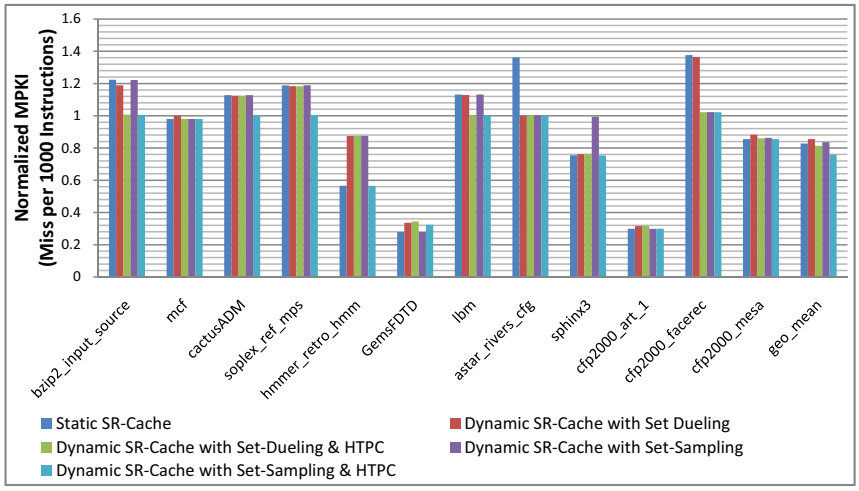

Figure 11: **The Effectiveness of Dynamic SR-Cache.** This figure gives the MPKI comparison of different Dynamic SR-Caches. The Dynamic SR-Cache using HTPC and *Set-Sampling* techniques produces the best performance. In all the benchmarks, its performance is quite close to the best between the performance of Static SR-Cache and that of the conventional cache using the LRU policy.

Cache running benchmark hmmer_retro_hmm, SR-Mode Sampler Sets have got totally 122411 accesses at one moment; in the meanwhile, ALRU-Mode Sampler Sets have got totally 180228 accesses. Although SR-Mode Sampler Sets have higher hit rate (92.1%) than that of ALRU-Mode (91.1%), Dynamic SR-Cache still chooses ALRU mode because ALRU-Mode Sampler Sets' access hit count is much larger than SR-Mode Sampler Sets' access hit count. Based on above analysis, we use *Set-Sampling*. *Set-Sampling* has *sampler* sets which is statically running one of the candidate policy, and each *sampler* set has an additional data-excluded copy (called *shadow* ways) running the other candidate policy (ALRU mode in Dynamic SR-Cache). *Set-Sampling* is different from *Set-Dueling* in that two candidate policies are facing the same access patterns, therefore it can avoid the inaccuracy in *Set-Dueling* technique. We can tell that the Dynamic SR-Cache using HTPC and *Set-Sampling* techniques has the best performance among differnt Dynamic SR-Caches (reducing 24.15 percent cache misses in geometric mean).

D. Comparison with State-of-the-Art Replacement Policies

We compare now the performance of a Dynamic SR-Cache with HTPC, which is the best of the proposed SR-Caches, with three state-of-the-art replacement policies (LRU, DIP[2] and DRRIP[5]). For fairness, all the dynamic cache policies (Dynamic SR-Cache, DIP, DRRIP) use *Set-Sampling* instead of *Set-Dueling*.

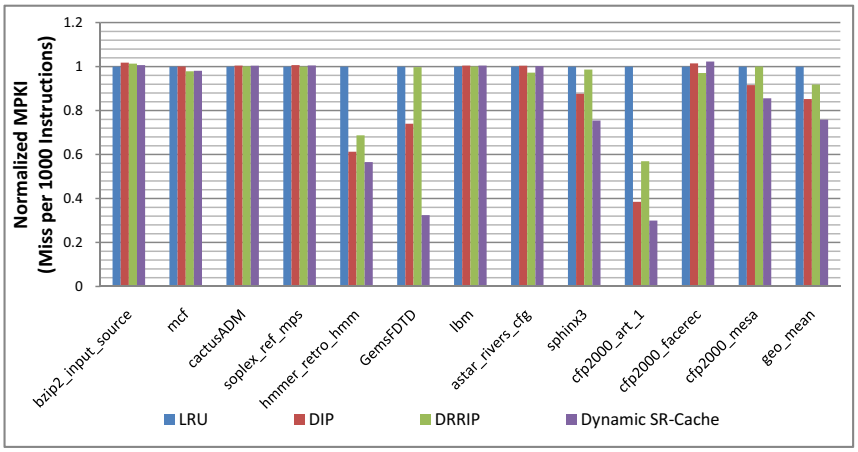

Figure 12: **Comparison of MPKI between High Threshold Dynamic SR-Cache and conventional caches with several state-of-the-art replacement policies.** We can see that the MPKI of High Threshold Dynamic SR-Cache is significantly lower than the conventional caches, no matter what replacement policies they adopt.

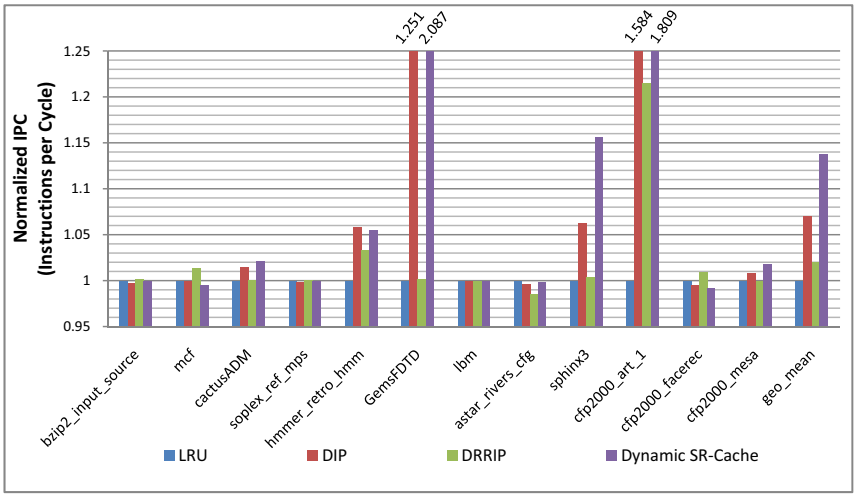

Figure 13: **Comparison of IPC between High Threshold Dynamic SR-Cache and conventional caches with several state-of-the-art replacement policies.** The improvement of IPCs in High Threshold Dynamic SR-Cache are the results of the MPKI drops.

As presented in Figures 12 and 13, for four benchmarks (GemsFDTD, sphinx3, cfp2000_art_1 and cfp2000_mesa), the Dynamic SR-Cache has a quite remarkable performance boost (up to 108.7% in benchmark GemsFDTD) compared to all other replacement policies. These benchmarks' LLC working set is larger than 1MB (the conventional last level cache's storage), and the access patterns are quite regular to predict. These attributes are desirable for the SR-Cache to explore the benefit of spontaneous reload. For another benchmark (hmmer_retro_hmm), the Dynamic SR-Cache still has quite a considerable performance gain compared to the LRU policy, and its performance increase is quite close to the DIP policy. For the rest of the benchmarks, though it does not have remarkable performance benefit, the Dynamic SR-Cache performs quite similarly to the baseline LRU policy cache. And for these benchmarks, neither DIP nor DRRIP policy have any remarkable performance gain compared to LRU policy.

E. Hardware Overhead

The main additional storage required is the overhead of Reuse History Field for every cache block. For each cache block, the additional *Reuse History Field* requires in total 10 bits, with 5 bits recording the *Idle Counts* and 5 bits storing the predicted *Reuse Interval*. Thus, for a Dynamic SR-Cache with 1024 sets and 16 ways, the additional storage requirement of *Reuse History Fields* is 20KB. If using *Set-Sampling* technique, some additional storage is required for shadow

ways in sampler sets. In our experiment configuration, Dynamic SR-Cache has 32 sampler set, and each sampler set has 16 shadow blocks running ALRU policy. Each shadow blocks takes up 37 bits (30 for address tag, 5 for *Idle Count Field*, 1 for valid bit and 1 for dirty bit), therefore all the 512 shadow blocks take up 2.3125KB. In conclusion, Dynamic SR-Cache requires addition storage of around 23KB, while a LRU policy conventional cache requiring 8KB of additional storage. Thus, the storage requirement of Dynamic SR-Cache is acceptable in view of the performance boost over conventional caches.

VII. Related Work

In recent years, due to growing concerns about the memory wall, the architectural design of LLCs has become a hot topic. The basic ideas of the previous work can be classified into two categories: cache management and prefetching.

1) Cache Management.: Cache management methods try to improve cache architecture, thus grant more possibility for useful data to stay in the cache. Much work has focused on different dimensions to improve the LLC architecture:

- *Dynamic Insertion Policy* (DIP) [2]: This paper is based on the fact that cache thrashing problem can be mitigated if some of the applications' workload stays in the cache. Thus, the authors propose *Bimodal Insertion Policy* (BIP) which can guarantee that data stay in the cache longer than they would with the LRU policy. They also proposes a low overhead mechanism to dynamically choose between two different replacement policies for a cache.
- *Instruction-based Reuse Distance Prediction* (IbRDP) [3]: Our work is partially inspired by this piece of work, which is the first to introduce the idea of using the *Reuse Distance* for cache replacement policies. They argue that, with the additional information of the instruction which triggers the cache access, it is possible to qualitatively predict the cache block reuse distances.
- *Protecting Distance Based Replacement Policy* (PDP) [13] : This is another recently proposed replacement policy based on *Reuse Distance*. This policy tries to protect a cache block long enough until it is reused.

And there are many other contributions [4], [5], [6], [15], [16], [17], [14] in this area. However, all these pieces of work treat LLC as a passive entity which can access memory only when there is a cache miss. Our SR-Cache is fundamentally different from these ideas in that SR-Cache is an active module which can spontaneously reload an off-chip data before it is accessed again.

2) Prefetching.: Prefetching tries to predict the accessed addresses in near future, and get the corresponding data into the cache prior to its access time. However, as a method which explores *spatial regularity* of access patterns, prefetching does not manage to exactly predict when the prefetched addresses will be accessed. Thus, a prefetching mechanism may deteriorate the cache thrashing problem by inserting data which will be used in a far-away future. Some work focuses on filtering out harmful prefetching using the cache's feedback information [8] [9] [10]. In a more recent proposal [11], more information of the cache is exposed to the prefetcher, and the prefetcher tries to insert the prefetched cache block into the different LRU stack position of the cache set. By doing this, the behavior of cache and prefetcher can be more coordinated.

Although it also loads off-chip data prior to its usage, spontaneous reloading technique is fundamentally different from prefetching in their purpose:

- Prefetching is a mechanism which discovers the *spatial regularity* of history accesses, and tries to use this regularity to predict future access addresses.
- Spontaneous reloading is a mechanism which discovers the *temporal regularity* of data blocks' accesses, and tries to actively reload back those data blocks which will be reused soon.

Thus, the SR-Cache does not compete with prefetchers. Instead, using the SR-Cache offers new possibility to solve prefetching pollution problems—when a block is evicted by a prefetching request in conventional caches, it loses all the history information; however, in the SR-Cache, the evicted block can preserve its history information in virtual block and be spontaneously reloaded back to the SR-Cache before it is accessed again.

VIII. Conclusion

This paper proposes a *Last Level Cache* (LLC) structure which uses virtual blocks to mimic a much larger cache. Our proposed mechanism requires minimal hardware overhead and incremental changes to a current cache architecture, but has a large performance advantage. This work makes following contributions:

- We propose an SR-Cache which can reload off-chip data into the cache before its reuse time. Different from prefetchers which focus on exploring *spatial regularity*, the SR-Cache targets the *temporal regularity* of access patterns.
- We also developed a Dynamic SR-Cache, which can manage to convert the LLC back to a LRU-like policy conventional cache when the SR-Cache mechanism does not perform well for the application currently running.

We plan additional work to further explore this idea, such as improving the accuracy of SR-Cache, coordinating the SR-Cache and prefetchers, and making the SR-Cache suitable for a shared cache.

ACKNOWLEDGMENTS

This work is in part supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302501, the National Science Foundation for Distinguished Young Scholars of China under Grant No. 60925009, the Foundation for Innovative Research Groups of the National Natural Science Foundation of China under Grant No. 60921002, National High-Tech Research & Development Program of China under Grant No. 2012AA012301, National Natural Science Foundation of China under Grant No. 61173007, 61100013, 61100015, 61204047, 61202059 and Beijing Science and Technology Plans under Grant No. 2010B058.

REFERENCES

[1] L Belady. A study of Replacement Algorithms for a Virtual-storage Computer. IBM Systems Journal, 1966.

[2] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. ISCA-34, 2007.

[3] Pavlos Petoumenos, Georgios Keramidas, and Stefanos Kaxiras. Instruction-based reuse-distance prediction for effective cache management. SAMOS-9, 2009.

[4] Mainak Chaudhuri. Pseudo-LIFO: the foundation of a new family of replacement policies for last-level caches. MICRO-42, 2009.

[5] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). ISCA-37, 2010.

[6] Moinuddin K. Qureshi, M. Aater Suleman, and Yale N. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. HPCA-13, 2007.

[7] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. IBM Technical White Paper, Oct. 2001.

[8] Xiaotong Zhuang and Hsien-Hsin S. Lee. 2007. Reducing Cache Pollution via Dynamic Data Prefetch Filtering. IEEE Trans. Comput. 56, 1 (January 2007), 18-31.

[9] V. Srinivasan, G. S. Tyson, and E. S. Davidson. A static filter for reducing prefetch traffic. Technical Report CSE-TR-400-99, University of Michigan Technical Report, 1999.

[10] W.-F. Lin, S. K. Reinhardt, D. Burger, and T. R. Puzak. Filtering superfluous prefetches using density vectors. In ICCD, 2001.

[11] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. HPCA-13, 2007.

[12] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. 2006. The M5 Simulator: Modeling Networked Systems. IEEE Micro 26, 4 (July 2006), 52-60.

[13] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota. 2012. Improving Cache Management Policies Using Dynamic Reuse Distances. MICRO-45, 2012.

[14] Dongyuan Zhan, Hong Jiang, Sharad C. Seth. 2010. Spatiotemporal Management of Capacity for Intra-Core Last Level Caches. MICRO-43, 2010.

[15] Pavlos Petoumenos, Georgios Keramidas, and Stefanos Kaxiras. 2007. Cache Replacement Based on Reuse-Distance Prediction. ICCD-25, 2007.

[16] Jayesh Gaur, Mainak Chaudhuri, and Sreenivas Subramoney. 2011. Bypass and Insertion Algorithms for Exclusive Last-Level Caches. ISCA-38, 2011.

[17] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely and Joel Emer. 2008. Adaptive Insertion Policies for Managing Shared Caches. PACT-17. 2008.