A Dueling Segmented LRU Replacement Algorithm with Adaptive Bypassing

Hongliang Gao Chris Wilkerson
Intel Corporation
{hongliang.gao, chris.wilkerson}@intel.com

Abstract

In this paper we present a high performance cache replacement algorithm called Dueling Segmented LRU replacement algorithm with adaptive Bypassing (DSB). The base algorithm is Segmented LRU (SLRU) replacement algorithm originally proposed for disk cache management. We introduce three enhancements to the base SLRU algorithm. First, a newly allocated line could be randomly promoted for better protection. Second, an aging algorithm is used to remove stale cache lines. Most importantly, we propose a novel scheme to track whether cache bypassing is effective. Based on the tracking results, we can adaptively adjust bypassing to fit workload behavior. DSB algorithm is implemented with a policy selector to dynamically select two variants of SLRU algorithms with different enhancements.

1. Introduction

This paper describes a Dueling Segmented LRU replacement algorithm with adaptive Bypassing, referred to DSB in the paper. The base Segmented LRU (SLRU) algorithm is described in Section 2. We propose two simple enhancements to SLRU algorithm described in Section 3. In Section 4, we discuss our proposed adaptive bypassing algorithm that prevents the allocation of cache lines under certain circumstances. DSB is implemented with a policy selector, described in Section 5, which enables DSB to adaptively select between two variants of SLRU depending on application behavior. In Section 6, we describe implementation cost and configurations of our algorithm. Section 7 presents performance results of the proposed algorithm.

2. Segmented LRU *Algorithm*

SLRU algorithm was originally proposed in [1] as a cache management algorithm for disk system. In this paper, we apply it as a processor cache replacement algorithm. The base SLRU algorithm augments each cache line with a reference bit dividing up the traditional LRU list of cache lines into two logical sub lists, the referenced list and the non-referenced list. The referenced-list consists of cache lines with the referenced bit set, and the non-referenced-list consists of the remainder of the cache lines. When choosing a line for replacement the LRU cache line in the non-referenced list is selected first. If all cache lines are in the referenced-list then we select the global LRU cache line from among all the cache lines in the set.

We propose three enhancements to the SLRU algorithm in this paper. By selecting which enhancements are used, we can get different variants of the SLRU algorithm. Our selected variants are described in Section 6.

3. SLRU with Random Promotion and Aging

Traditional implementations of SLRU mark the reference bit when a cache line is referenced, but previous work has shown benefit to making selected random promotions as well [2]. The random promotion in the SLRU algorithm allows the reference bit to be randomly set for newly allocated cache lines. For example, if promotion probability is 1/32, then 1 in 32 newly allocated lines will be promoted by marking its reference bit.

Another enhancement to SLRU is an aging mechanism to allow lines in the referenced list to be aged or removed from the reference list. When a cache line is allocated the reference bit of the LRU element is cleared. We refer to the process of clearing the reference bit as aging.

4. SLRU with Adaptive Bypassing

The third enhancement to SLRU is our adaptive bypassing algorithm which prevents the

allocation of lines in the cache under certain conditions. Although we only apply it to SLRU in this paper, it can be coupled with other replacement algorithms as well. Our experiment results have shown that it is also helpful for various other replacement algorithms.

The decision on whether or not to bypass the cache is made by bypass tracking logic implemented for each set in the cache. The bypassing logic evaluates the impact that a bypassing decision would have on the performance of the cache.

On each cache miss, we can choose to bypass the cache and keep the selected victim or we can replace the victim with the new line. The selected victim and new line are essentially competitors for the available cache line. Our scheme allows us to measure which of these two lines is more useful and to adjust the bypass algorithm accordingly.

When a line is selected to bypass the cache, we would like to assess the impact that the ***decision to bypass*** the line will have on the hit rate of the set. The decision to bypass is evaluated by storing the tag of the bypassed line in a field we refer to as *competitor tag*. In addition, we store a pointer to the tag of the victim line referred as *competitor way*. It is the line that would have been replaced without bypassing. Subsequent references to the set are compared against the tag of the bypassed line and the tag of the competitor way. We interpret a reference to the competitor way before the bypassed line as an indication of an effective bypass. Conversely, a reference to the bypassed line before the competitor way is treated as an indication of an ineffective bypass. After we detect whether a bypass is effective/ineffective, we increase/decrease the bypassing probability.

In some cases we may make a ***decision not to bypass*** a given cache line. In order to assess the impact of bypassing in this case, we randomly select some newly allocated lines as "*virtually bypassing*" the cache. The line is still allocated in the cache. Instead, a decision is made to evict the victim line. The decision not to bypass is evaluated in much the same way as the decision to bypass. A pointer to the tag of the allocated line, the virtually bypassed line in this case, is stored as well as a copy of the tag of the evicted line, the victim line in this case. The references to each of these are tracked and treated similarly as described for the decision to bypass.

Our bypass algorithm randomly selects newly allocated cache lines for bypassing. The probability of making the bypass is determined by how effective the decisions to bypass have been in the past as described above. Each effective bypass doubles the probability that a future a bypass will occur, for example, if the current probability is 0.25 the probability will double to 0.5. Similarly each ineffective bypass halves the probability of a future bypass, for example, cutting the current probability of 0.5 to 0.25. For virtual bypassing, we use a fixed probability.

The minimum bypassing probability is set to 0 that will prevent any bypassing and allocate all missed lines. Note that even if the probability is adjusted to 0, we can still track effectiveness of bypassing through virtual bypassing, which could turn on bypassing later if program behavior changes to favor bypassing. The maximum bypassing probability is set to 1, which will essentially bypass all cache misses until the tracking scheme detects ineffective bypassing decisions.

5. Policy Selector

Our design allows 2 dueling replacement algorithms to be evaluated concurrently, and uses a policy selector to select between the 2 algorithms. For example, traditional LRU and SLRU replacement polices could be evaluated and the policy selector will attempt to select the policy that is most appropriate for the application that is being run.

The policy selector uses set sampling [2][3]. The sampled sets are chosen statically by the algorithm proposed in [3] and do not vary during the execution of the workload. We augment each of these sets with an *auxiliary tag directory* (ATD) [3] to concurrently implement both dueling policies (policy A, policy B) for these sets. In addition, we use a saturating counter (policy selector counter) to track the performance of policy A & B in the sampled sets.

Each reference to the sampled sets will result in an update to the saturating counter only if policy A & B yield different results. For example, if policy A & B are either both hit or miss, the policy selector counter is not updated.

On the other hand, if policy A is a miss and policy B is a hit, the policy selector counter is incremented. Conversely, if policy A produces a hit but policy B misses the policy selector counter is decremented.

6. Hardware Cost and Complexity

For a 16-way associative cache, the base SLRU algorithm requires 5 bits per cache line to implement. Four bits are used to implement the true LRU stack and 1 bit is used to mark lines belonging to the referenced list.

Aging do not require extra storage in the cache. Random promotion shares the random generator used in adaptive bypassing logic.

The adaptive bypassing logic requires 22 bits per cache set. First, the competitor tag is used to store the lower 16 tag bits of the tracked line. The competitor way needs 4 bits in a 16-way cache. There are two other 1-bit fields to record the status of tracking. *Competitor valid* bit is used to mark whether the competitor way is still valid. We set this bit when start tracking a bypassing and clear it when the competitor way is selected as the victim or we have reached a decision of whether the bypassing is effective or ineffective. *Virtual bypassing* bit is the indication of whether current tracked bypassing is a result of real or virtual bypassing.

The storage cost of policy selector is mainly consumed by ATD. In our configuration, the ATD includes a sampled set to track for every 32 cache sets. Each sampled set has separated tags and replacement states for each policy. Similar as earlier proposals for dynamic policy selection [4], we use 16-bit partial tags in ATD. So each set has (16-bit tag + 1-bit valid) * 16 ways * 2 policies = 544 bits. To support two variants of SLRU algorithms, we need storage for replacement state in each set for both policies. Depending on whether bypassing is enabled, the cost of replacement state is 80 bits (without adaptive bypassing) or 102 bits (with adaptive bypassing) per set for each policy.

Besides hardware cost mentioned above, there are several registers used for adaptation. An 11 bit saturating counter is used for policy selection. The bypassing probability is determined by a 4-bit register per policy to select a probability among 14 predetermined choices: 0 and $1/2^K$ (K ranges from 0 to 12). Random numbers are needed for random promotion and adaptive bypassing. In this paper, we use a 32-bit Linear Feedback Shift Register (LFSR) as the random number generator.

In our submission to the 1st JILP Cache Replacement Championship (CRC1) [5], we include three representative configurations of the DSB algorithm as described in Table 1. All configurations are composed with two variants of SLRU algorithms and adaptive bypassing support.

Table 1. DSB Configurations

	CONFIG 1	CONFIG 2	CONFIG 3
Enable bypassing for policy0	True	True	True
Enable bypassing for policy1	False	True	True
Random promotion probability for policy0	0	0	0
Random promotion probability for policy1	0	0	16
Aging for policy0	0	0	0
Aging for policy1	1	1	1
Virtual bypassing probability	16	8	8
Initial bypassing probability	64	64	8
Second minimum bypassing probability (minimum is 0)	1/256	1/4096	1/4096

The total hardware cost of CONFIG3 is summarized in Table 2. CONFIG2 has the same cost as CONFIG3. CONFIG1 has smaller cost since bypassing is disabled in one policy so that only one policy in ATD requires extra storage to support bypassing. As shown in Table 2, the total storage cost of our proposed algorithm is less than CRC1's budget for both single thread and multi-thread tracks.

Table 2. Hardware Cost (bits)

	Single-core track	Multi-core track
# of cache sets	1k	4k
# bits for SLRU per cache line	5	5
# bits for bypassing per cache set	22	22
Total cost of a set	**5*16+22 = 102**	**5*16+22=102**
# bits per ATD set	544+102*2 = 748	544+102*2 = 748
# of sets in ATD	32	128
Total cost of ATD	**748*32=23.4k**	**748*128=93.5k**
LFSR	32	32
Policy selection counter	11	11
Bypassing probability register	4*2 = 8	4*2 = 8
Total DSB Cost	**102*1k+23.4k+ 32+11+8 = 125.4K**	**102*4k+93.5k+ 32+11+8 = 501.5k**

7. Results

In order to cover a large variant of program behaviors, we tuned our algorithm with 47 SPEC CPU2006 traces. These traces are picked based on their sensitivity to different cache replacement algorithms from 148 SPEC CPU 2006 traces with different inputs and simulation points. Due to the limited space, we only report results of 15 of these traces in this section. These 15 traces are selected based on the criteria that more than 1% performance could be achieved when increase LLC capacity from 1M to 2M with true LRU replacement algorithm. These traces are simulated for 100M instructions after skipping 40B instructions.

Using the DSB configurations shown in Table 1, LLC miss rates (measured in MPKI reduction compared to true LRU) are shown in Figure 1. Figure 2 gives performance speedups of the selected traces with true LRU as the baseline. As shown in the figures, the proposed algorithm achieves significant higher performance than true LRU. Among the 15 traces, only two traces (*zeusmp* and *soplex*) show performance degradation and the maximum degradation is relatively small (-2%). The overall performance improvements are 7.7%, 8.6% and 8.4% respectively for CONFIG1, CONFIG2 and CONFIG3.

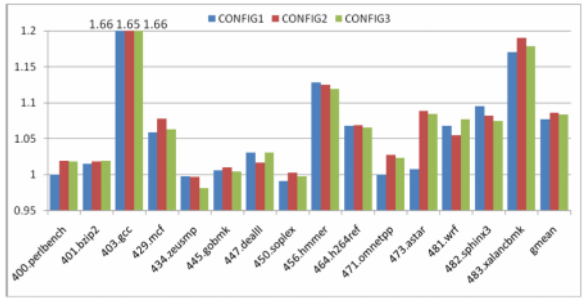

Figure 1. MPKI reductions compared to true LRU

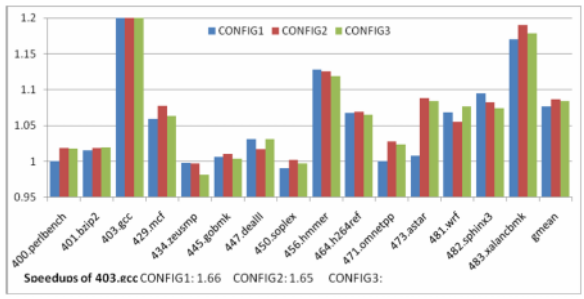

Figure 2. Speedups with true LRU as the baseline

Among the three enhancements proposed in this paper, adaptive bypassing contributes to majority of the performance gain over true LRU. Without bypassing, performance gains of all three configurations drop to about 2.7%. We also observe significant performance improvement from adaptive bypassing in single policy algorithms. For example, a true LRU algorithm with adaptive bypassing has 6.9% performance gain.

8. References

[1] R. Karedla et al, "Caching Strategies to Improve Disk System Performance", Computer, vol. 27, no. 3, pp. 38-46, Mar. 1994.

[2] M. Qureshi et al, "Adaptive Insertion Policies for High Performance Caching", ISCA 34, June 2007.

[3] M. Qureshi et al, "A case for MLP-aware cache replacement", In ISCA-33, 2006.

[4] R. Subramanian et al, "Adaptive Caches: Effective Shaping of Cache Behavior to Workloads", in MICRO-39, 2006.

[5] 1st JILP Workshop on Computer Architecture Competitions (JWAC-1): Cache Replacement Championship, http://www.jilp.org/jwac-1/.