IEICE TRANS. INF. & SYST., VOL.E96–D, NO.2 FEBRUARY 2013

LETTER

Bypass Extended Stack Processing for Anti-Thrashing Replacement in Shared Last Level Cache of Chip Multiprocessors*

Young-Sik EOM[†], **Jong Wook KWAK**[††a)], **Seong-Tae JHANG**[†††], ***Nonmembers***, ***and*** **Chu-Shik JHON**[†], ***Member***

SUMMARY Chip Multiprocessors (CMPs) allow different applications to share LLC (Last Level Cache). Since each application has different cache capacity demand, LLC capacity should be partitioned in accordance with the demands. Existing partitioning algorithms estimate the capacity demand of each core by stack processing considering the LRU (Least Recently Used) replacement policy only. However, anti-thrashing replacement algorithms like BIP (Binary Insertion Policy) and BIP-Bypass emerged to overcome the thrashing problem of LRU replacement policy in a working set greater than the available cache size. Since existing stack processing cannot estimate the capacity demand with anti-thrashing replacement policy, partitioning algorithms also cannot partition cache space with anti-thrashing replacement policy. In this letter, we prove that BIP replacement policy is not feasible to stack processing but BIP-bypass is. We modify stack processing to accommodate BIP-Bypass. In addition, we propose the pipelined hardware of modified stack processing. With this hardware, we can get the success function of the various capacities with anti-thrashing replacement policy and assess the cache capacity of shared cache adequate to each core in real time.

key words: *last level cache, stack processing, replacement policy, anti-thrashing, cache partitioning, chip multi-processors*

1. Introduction

Recently, the pressure on the memory system increases due to the widening speed gap of processor and memory and the increasing number of cores. One of the keys to obtain high performance is to manage LLC (Last Level Cache) efficiently so that off-chip accesses are reduced.

Since each application has different cache capacity demand, LLC capacity should be partitioned in accordance with application demands. UCP (Utility-aware Cache Partitioning) [3] estimates the cache utility of each core which is the increased number of cache hits as cache capacity increases through stack processing [1]. Then, they assign cache ways to each core to obtain the maximum number of hits. It uses only the LRU (Least Recently Used) replacement policy.

The LRU replacement policy has the advantage of good performance for high locality workloads. However, it can show thrashing behavior for a working set greater than the available cache size. To alleviate thrashing problem, BIP (Bimode Insertion Policy) [4] inserts most of new cache blocks to the LRU position to preserve the cache contents and inserts the rest of new blocks to the MRU position to adapt to working set changes. BIP-Bypass [4] bypasses new blocks instead of inserting them to LRU position.

In TADIP [2], each core determines whether or not it use LRU or BIP as a replacement policy to attack thrashing problem in CMP. However, TADIP cannot partition the shared cache and each core cannot preserve its working set in cache space.

Until now, there is no cache partitioning method with anti-thrashing replacement policy, since there is no stack processing method for anti-thrashing replacement policy. In this letter, we prove that BIP is not feasible to stack processing but BIP-Bypass is. In addition, we modify stack processing to accommodate BIP-Bypass, and we propose new hardware to implement stack processing for BIP-Bypass.

2. Stack Processing

2.1 Inclusion Property, Stack and Success Function

Let x_t be the address of a trace in time t. *Inclusion property* means that the cache contents $B_t(C)$ must be a subset of $B_t(C+1)$ for any time t and any capacity C on a trace as follows.

$$B_t(1) \subset B_t(2) \subset B_t(3) \cdots$$

Stack algorithms are replacement algorithms that satisfy inclusion property [1].

From the inclusion property, the cache contents for all capacities can be represented in the following way. Stack is an ordered address list $S_t = S_t(1), S_t(2), S_t(3), \ldots$, where

$$S_t(i) = B_t(i) - B_t(i-1) \qquad \text{for } i = 1, 2, \ldots$$

This can be used to efficiently determine the *success function* $F(C)$. Let C_t denote the least buffer capacity such that $x_t \in B_{t-1}(C)$. C_t is called *critical capacity* because all buffers larger than C_t contains x_t from inclusion property. C_t is simply the position of page x_t in the stack S_{t-1}. This position is called *stack distance* Δ_t.

Let $n(\Delta)$ denote the number of times the stack distance Δ is observed in processing a trace. Since stack distance

Manuscript received September 13, 2012.

[†]The authors are with School of EECS, Seoul National University, Korea.

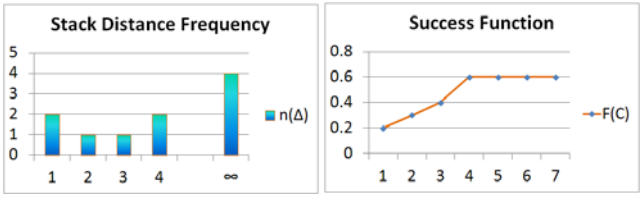
[††]The author is with the Department of Computer Engineering, Yeungnam University, Korea.

[†††]The author is with the Department of Computer Science, The University of Suwon, Korea.

*This work was supported by the 2012 Yeungnam University Research Grant and by GRRC program of Gyeonggi province [(GRRC SUWON2012-B1) Cooperative CCTV Image Based Context-Aware Process Technology].

a) E-mail: kwak@ynu.ac.kr

DOI: 10.1587/transinf.E96.D.370



(a) (b)

Fig. 1 An example of stack distance frequency and success function.

equals critical capacity, the number of times that the referenced address is found in the cache with capacity C is

$$N(C) = \sum_{\Delta=1}^{C} n(\Delta)$$

and success function is given by the expression where L is the length of the trace

$$F(C) = N(C)/L.$$

$n(\Delta)$ can be determined from a set of distance counters. All counters are set initially to zero, and the counter for each distance Δ is incremented whenever that distance occurs.

Figure 1 shows the example of the stack distance frequency function and the success function after the trace of 10 addresses. To find the number of hits for the buffer capacity 3, $N(3) = n(1) + n(2) + n(3) = 4$ as indicated in Fig. 1 (b). In this case, hit rate is $F(3) = N(3)/10 = 0.4$.

2.2 Total Ordering

Replacement algorithms that induce a total ordering on all previous referenced addresses and use this ordering to make replacement decisions satisfy inclusion property [1]. Two representative replacement policies satisfying total ordering are LRU and LFU (Least Frequently Used) with a break tie scheme. They maintain just one priority list independent on the cache capacity. When there is a new block to insert, they choose the lowest priority block in the cache for replacement. Replacement policies that do not hold total ordering like FIFO should maintain one priority list per each cache capacity. Therefore, replacement policies with total ordering are more area efficient when we want to evaluate various cache capacities.

3. Correlation between Insertion and Promotion

Recent replacement policies consider insertion policy and promotion policy independently. BIP [4] inserts new blocks to LRU position or MRU position and promotes hit blocks to the MRU position. PIPP [5] inserts new blocks to the position assigned for each core and promotes one step toward MRU position in case of hit.

Figure 2 (a) shows the cache contents with size 1 to 5 after 'a', 'b', 'c', 'd', 'e' reference with MRU insertion policy. Note that inclusion property and total ordering hold. Cache contents are ordered by priority list.

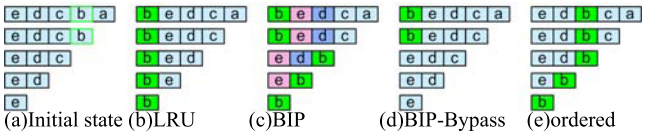


Fig. 2 An example of cache contents.

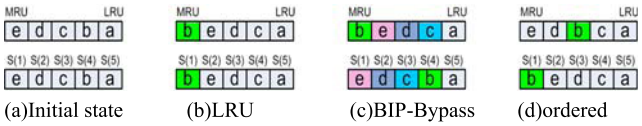


Fig. 3 An example of LRU list and stack.

Figure 2 (b), 2 (c) and 2 (d) show the cache contents and priority list state changed by LRU, BIP and BIP-Bypass respectively after new reference 'b'. In Fig. 2 (b), LRU promotes and inserts to MRU position. It does not break total ordering. However, LRU is not anti-thrashing replacement policy. In Fig. 2 (c), BIP promotes 'b' to MRU position at hit and inserts 'b' to LRU position at miss. In this case, the orders of (b, e) and (b, d) in the caches with capacity 4 and 5 are different from the orders in the cache with capacity 2 and 3. These discrepancies in ordering list break total ordering and make a victim selection with single priority list impossible. This is why BIP is not feasible to stack processing.

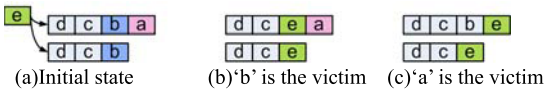
Promotion policy and insertion policy are dependent on each other. In Fig. 2 (e), when 'b' is inserted to the LRU position in the cache with capacity 3, 'b' cannot be promoted over 'd' position in the cache with capacity 4 and 5 to satisfy total ordering. When 'b' is promoted one step toward the MRU position in the caches with capacity 4 and 5, 'b' should be inserted to the LRU position in the caches with capacity 2 and 3. Only MRU promotion and MRU insertion policy have no ordering constraint because they promote and insert blocks to the same position regardless of hits or misses.

In Fig. 2 (d), BIP-Bypass promotes 'b' to the MRU position in the cache with capacity 4 and 5 and bypasses 'b' in the cache with capacity 1, 2 and 3. Since BIP-Bypass holds total ordering, it is suited for efficient stack processing. When using MRU insertion policy, BIP-Bypass must also maintain MRU-promotion policy to keep the total ordering.

Figure 3 (a), 3 (b), 3 (c) and 3 (d) shows the LRU priority list and the stack of size 5 cache corresponding to Fig. 2 (a), 2 (b), 2 (d) and 2 (e), respectively. Figure 3 (c) shows that BIP-Bypass updates LRU priority list and stack differently. Therefore, BIP-Bypass must maintain two lists (LRU priority list and stack). Unlike BIP-Bypass, stack processing for LRU replacement policy needs only one list, since LRU priority list is equivalent to stack at any time as in Fig. 3 (b).

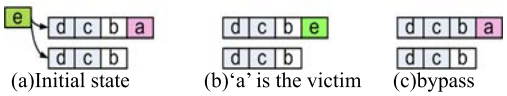
4. Bypass Extended Stack Processing

Conventional stack processing is based on the demand pag-



(a)Initial state (b)'b' is the victim (c)'a' is the victim

Fig. 4 An example of the insertion 'e'.



(a)Initial state (b)'a' is the victim (c)bypass

Fig. 5 An example of the bypass.

ing which means that only newly referenced addresses should be inserted to the cache. Since BIP-Bypass allows new blocks to bypass, stack processing also should be modified to accommodate bypass.

4.1 Inclusion Condition

$Y_t(C)$ represents the replacement victim of $B_{t-1}(C)$ at time t. "$Y_t(C)$ is $\emptyset$" means that new block is bypassed and there is no victim for replacement.

If inclusion property is satisfied up to and including time $t-1$, and there is a miss on $C+1$ capacity, inclusion property is satisfied if and only if inclusion condition is satisfied at time t. **Inclusion Condition** is as follows.

1) If $Y_t(C)$ is not $\emptyset$, $Y_t(C+1)$ is $Y_t(C)$ or $S_{t-1}(C+1)$.

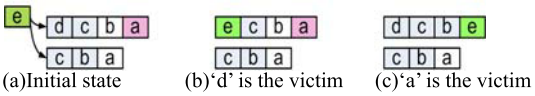
2) If $Y_t(C)$ is $\emptyset$, $Y_t(C+1)$ is $\emptyset$ or $S_{t-1}(C+1)$.

The proof of necessary condition for inclusion property is as follows. 1) When $Y_t(C)$ is not $\emptyset$, if $Y_t(C+1)$ is neither $Y_t(C)$ nor $S_{t-1}(C+1)$, $Y_t(C+1)$ is an other block z or $\emptyset$. If $Y_t(C+1)$ is z, z is included in $B_t(C)$, but not in $B_t(C+1)$, which violates inclusion property. If $Y_t(C+1)$ is $\emptyset$, new block violates inclusion property. 2) When $Y_t(C)$ is $\emptyset$, if $Y_t(C+1)$ is neither $\emptyset$ nor $S_{t-1}(C+1)$, $Y_t(C+1)$ is an other block z. z is included in $B_t(C)$, but not in $B_t(C+1)$, which violates inclusion property.

Inclusion condition is also sufficient, because if inclusion condition holds, $B_t(C)$ is a subset of $B_t(C+1)$.

Figure 4 is an example of the insertion 'e' to caches with size 3 and 4. Figure 4 (a) is the initial state. When 'e' is inserted to the both caches, inclusion property is satisfied when (Fig. 4 (b)) 'b' or (Fig. 4 (c)) 'a' is the victim of cache with size 4. Choosing another block like 'd' (or 'c') violates inclusion property since 'd' (or 'c') is in the cache with size 3, but is not in the cache with size 4. We can see that among $Y_t(C)$ and $S_{t-1}(C+1)$, one becomes $Y_t(C+1)$ and the other becomes $S_t(C+1)$.

Figure 5 is an example of a bypass 'e' to the cache with size 3. Figure 5 (a) is an initial state. When 'e' is bypassed by the cache with size 3, inclusion property is satisfied when (Fig. 5 (b)) 'a' is the victim of the cache with size 4 or (Fig. 5 (c)) the cache with size 4 bypasses 'e'. Choosing another block violates inclusion property. From inclusion condition, we can see that only when $B_{t-1}(C)$ bypasses new block, $B_{t-1}(C+1)$ can bypass.



(a)Initial state (b)'d' is the victim (c)'a' is the victim

Fig. 6 The inclusion property violation of total ordering.

4.2 Total Ordering Inclusion Condition

Just total ordering alone does not satisfy inclusion property in the presence of bypass. We define $min(A)$ as the lowest priority block in A, and $max(A)$ as the highest priority block in A.

From inclusion condition, if $Y_t(C)$ is $\emptyset$, $Y_t(C+1)$ must be $\emptyset$ or $S_{t-1}(C+1)$. If $B_{t-1}(C+1)$ wants to insert a new block, it can violate inclusion property since priority list can choose a victim other than $S_{t-1}(C+1)$. Total ordering priority list chooses $min[B_{t-1}(C+1)]$ as a victim, but it cannot guarantee that $min[B_{t-1}(C+1)]$ is $S_{t-1}(C+1)$. Therefore, $\emptyset$ is the only option for $Y_{t-1}(C+1)$. To preserve the inclusion property, if $Y_t(C)$ is $\emptyset$, $Y_t(C+1)$ must be also $\emptyset$. In other words, when we use priority list to make replacement decisions, if all $Y_t(C)$ for every capacity C with miss is 1) $\emptyset$ or 2) not $\emptyset$, that is **Total Ordering Inclusion Condition**, inclusion property is satisfied.

The proof of total ordering inclusion condition is as follows.

If all $Y_t(C)$ is not $\emptyset$,

$$
\begin{aligned}
&Y_t(C) = min[B_{t-1}(C)] \text{ and}\\
&Y_t(C+1) = min[B_{t-1}(C+1)]\\
&\qquad = min[B_{t-1}(C), S_{t-1}(C+1)]\\
&\qquad = min[min[B_{t-1}(C)], S_{t-1}(C+1)]\\
&\qquad = min[Y_t(C), S_{t-1}(C+1)]
\end{aligned}
$$

From inclusion condition, inclusion property holds. If all $Y_t(C)$ is $\emptyset$, inclusion property holds since all the cache contents do not change.

Figure 6 shows the inclusion property violation when we choose a victim by total ordering list without total ordering inclusion condition. Figure 6 (a) is the initial state after the last reference 'd' is bypassed in the cache with size 3 and inserted to the cache with size 4. Assume that a new reference 'e' occurs and the cache with size 3 bypasses 'e'. In Fig. 6 (b), if we insert 'e' to the cache with size 4, 'd' should be replaced to preserve the inclusion property. However, 'e' is not the victim which total ordering priority list determines. In Fig. 6 (c), since 'a' is the lowest priority block, 'a' is replaced instead of 'd' and inclusion property is violated. To avoid this violation, cache with size 4 should also bypass 'e'.

4.3 Stack Update

At every reference, we need to update stack to maintain up-to-date state as follows.

If $Y_t(C)$ is not $\emptyset$, by inclusion condition,

$S_t(C+1). = max[Y_t(C), S_{t-1}(C+1)]$.

If $Y_t(C)$ is $\emptyset$, $Y_t(C+1)$ is also $\emptyset$ by total ordering inclusion condition. Since the contents of $B_{t-1}(C)$ and $B_{t-1}(C+1)$ do not change,

$S_t(C+1) = S_{t-1}(C+1)$.

5. Hardware for Bypass Extended Stack Processing

To evaluate the success function of various cache capacities with BIP-Bypass replacement policy in real time, we need the hardware implementation of bypass extended stack processing.

In this implementation, pipelined architecture is natural, since $S_t(C)$ and $Y_t(C)$ values of cache capacity C are dependent on the values of capacity $C-1$. In addition, it also provides high bandwidth and scalability. Priority list is also pipelined to match pipelined stack.

Figure 7 shows the conceptual hardware structure of priority list and stack. Since all stages of stack need the priority value of new block, the pipelined priority list is accessed first, and then the pipelined stack is accessed second. Each stage of stack augments *pri* field which represents the priority of stack entry. In our implementation, larger *pri* value means lower priority.

Figure 7 also shows the example of *pri* update. Figure 7 (a) is the initial state of the LRU list and stack, corresponding to Fig. 2 (a). In Fig. 7 (b), after new reference '*b*', it is promoted to the MRU position in priority list. To reflect the updated LRU list to *pri* field, *pri* values under the position of '*b*' equal to 4 are increased by one and *pri* value of '*b*' is set to one along the stages. The counter of stack '*b*' is incremented by one to calculate the number of the references of stack distance 4.

The *N*th stage of the pipelined priority list means the *N*th entry from MRU position of priority list. *New ref* is newly referenced address, and *prev* and *addr* are the addresses of $N-1$ and N stages. At the first stage, *New ref* becomes *addr* to emulate the promotion to the MRU position. Figure 8 shows the *N*th stage of the pipelined priority list. In the purple box, if hit occurred in one of the previous stages, *prev* becomes *addr*. This is identical to shifting previous stage to current stage. In the green box, if *addr* is equal to *New ref*, current stage number N becomes the priority of *New ref*. Further stages just pass the priority to the next stage. If each stage augments a counter for stack distance, this pipelined priority list can be used for stack for the LRU replacement policy.

Figure 9 shows $C+1$ stage of the pipelined stack. $Pri(A)$ is the priority value of A in stack. With the found priority value at the priority list, the first stack stage starts.

Stage $C+1$ represents the $C+1$th stack entry. It calculates $S_t(C+1)$ and $Y_t(C+1)$ from $S_{t-1}(C+1)$ and $Y_t(C)$ and turns $Y_t(C+1)$ over stage $C+2$. The green box updates the counter which represents the number of references of $C+1$ stack distance. *New ref* is compared to $S_{t-1}(C+1)$. If they match, the counter for $S(C+1)$ is increased by one. The blue box updates stack. It operates only when new block is not bypassed. From inclusion condition and stack update, if $Pri[S_{t-1}(C+1)]$ is bigger than $Pri[Y_t(C)]$, $S_{t-1}(C+1)$ becomes $Y_{t+1}(C+1)$ and $Y_t(C)$ becomes $S_t(C+1)$. Otherwise, $S_{t-1}(C+1)$ becomes $S_t(C+1)$ and $Y_t(C)$ becomes $Y_{t+1}(C+1)$. The purple box adjusts *pri* field. If $Pri[S_t(C+1)]$ is smaller than $Pri[New\ ref]$, $Pri[S_t(C+1)]$ is increased by one. If hit occurs in this stage, one is assigned to $Pri[S_t(C+1)]$.

First stage is slightly different from other stages. It contains bypass decision logic which decides whether or not it bypasses the newly referenced block. It generates bypass signal randomly with high probability to preserve cache contents.

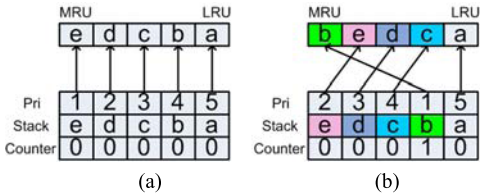


Fig. 7 The conceptual hardware structure of priority list and stack.

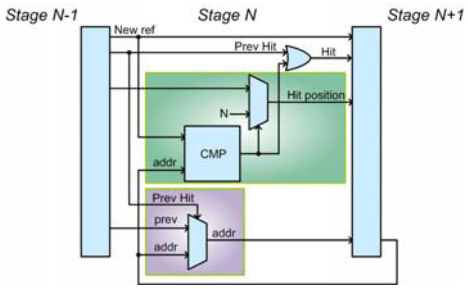


Fig. 8 *N*th stage of the pipelined priority list.

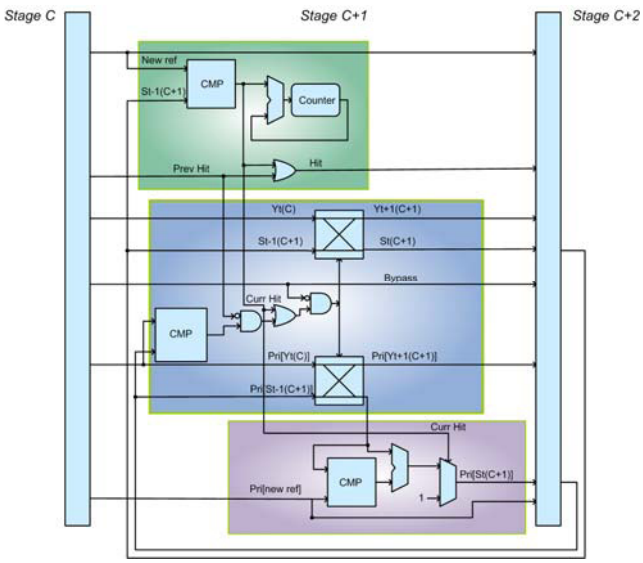


Fig. 9 $C+1$th stage of the pipelined stack.

6. Conclusion

BIP and BIP-Bypass are anti-thrashing replacement policies. In this letter, we proved that BIP replacement policy is not feasible to stack processing but BIP-bypass is. We modified stack processing to accommodate bypass, and we proposed the pipelined hardware architecture of stack processing for BIP-Bypass. With this hardware, we can assess the capacity demand of each core with anti-thrashing replacement policy in real time. Using this information, shared cache can be partitioned by anti-thrashing replacement policy as well as LRU replacement policy.

References

[1] J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation techniques for storage hierarchies," IBM System Journal, vol.9, no.2, pp.78–117, 1970.

[2] A. Jaleel, W. Hasenplaugh, M.K. Qureshi, J. Sebot, S.C. Steely Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," In PACT, 2008.

[3] M.K. Qureshi and Y.N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," Proc. MICRO, 2006.

[4] M.K. Qureshi, A. Jaleel, Y.N. Patt, S.C. Steely Jr., and J. Emer, "Adaptive insertion policies for high performance caching," pp.381–391, In ISCA-2007.

[5] Y. Xie and G.H. Loh, "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches," In ISCA-2009.