IPSJ Transactions on System LSI Design Methodology Vol. 4 166–181 (Aug. 2011)

Regular Paper

Exact, Fast and Flexible L1 Cache Configuration Simulation for Embedded Systems

Masashi Tawada,[†1] Masao Yanagisawa,[†2]
Tatsuo Ohtsuki[†1] and Nozomu Togawa[†1]

Since target applications running on an embedded processor are much limited in embedded systems, we can optimize its cache configuration based on the number of sets, block size, and associativities. An extremely fast cache configuration simulation method, CRCB (Configuration Reduction approach by the Cache Behavior), has been recently proposed which can calculate cache hit/miss counts accurately for possible cache configurations when the three parameters above are changed. The CRCB method assumes LRU-based (Least Recently Used-based) cache but many recent processors use FIFO-based (First In First Out-based) cache or PLRU-based (Pseudo LRU-based) cache due to its hardware cost. In this paper, we propose exact and fast L1 cache configuration simulation algorithms for embedded applications that use PLRU or FIFO as a cache replacement policy. Firstly, we prove that the CRCB method can be applied not only to LRU but also to other cache replacement policies including FIFO and PLRU. Secondly, we prove several properties for FIFO- and PLRU-based caches and we propose associated cache simulation algorithms which can simulate simultaneously more than one cache configurations with different cache associativities accurately for FIFO or PLRU. Finally, many experimental results demonstrate that our cache configuration simulation algorithms obtain accurate cache hit/miss counts and run up to 249 times faster than a conventional cache simulator.

1. Introduction

Recently, memory access speed has much increased but does not catch up with the processing performance in embedded processors, i.e., memory access speed limits overall the processing performance. By introducing a cache and constructing a memory hierarchy, we can bridge the gap between the processing performance and the memory access speed.

Cache memory access speed cannot increase as expected unless the cache is properly configured. It cannot reach to the required speed if we use a too small cache configuration. It will have extra power consumption and extra cost if we use a too much large cache configuration. Since application programs running on embedded processors are limited unlike those on general-purpose processors, embedded processors can be optimized for a particular application program. In particular, cache configuration can be optimized in terms of speed and power. Counting cache hits/misses is the important key to configure such an application-driven cache accordingly.

Cache configuration is parameterized by the number of sets, block size, and associativity[⋆1]. Cache configuration is also characterized by its cache replacement policy, that is an algorithm when and where a datum is stored to and discarded from a cache set. Practical processors use LRU-based (Least Recently Used-based) cache, FIFO-based (First In First Out-based) cache, and PLRU-based (Pseudo LRU-based) cache. If a processor uses an LRU cache whose associativity is large, its hardware cost increases too much. On the other hand, FIFO and PLRU can be implemented with low hardware cost even if its associativity is large. Many practical processors, such as PowerPC PPC755 and x86 processors, use FIFO or PLRU rather than LRU.

There are roughly two types of cache hit/miss measuring methods: The first ones just calculate and estimate cache hit/miss counts based on cache behavior [4),11)]. The second ones simulate memory accesses and obtain cache hit/miss counts accurately [8),9),12),13)]. The first ones run very fast but may have large errors in cache hit/miss counts. The second ones are accurate but time consuming. This paper targets the second approach, a simulation-based approach. In simulation-based cache hit/miss measuring, we take the following approach: When an application program runs on a single-core in-order processor, memory

†1 Department of Computer Science and Engineering, Waseda University
†2 Department of Electronic and Photonic Systems, Waseda University

⋆1 Our target cache here is a *single-level* cache configuration parameterized by the number of sets, block size, and associativity. In this sense, our proposed algorithm here cannot directly be applied to a multi-level cache which includes cache parameters more than a single-level cache. But, even if a multi-level cache, such as L1/L2-cache and L1/L2/L3-cache, is given, our proposed algorithm can be applied to each one of the cache levels. For example, our proposed algorithm can be applied to L2 cache configuration simulation if L1 cache configuration is completely fixed in L1/L2-cache.

access sequence from the processor is independent of a particular cache configuration. This means that cache configurations can be independent of memory access history to a main memory. Using this memory access history, we can simulate cache hit/miss counts in various cache configurations. However, simulating cache hits/misses takes too much time and boosting up the cache configuration simulation is the most important key there.

Several cache configuration simulation methods have been reported. Particularly, the CRCB (Configuration Reduction approach by the Cache Behavior) method[13] runs 205 times faster than a naive exhaustive approach. However it assumes only LRU as a cache replacement policy and it is not proved that it can be applied to FIFO and/or PLRU.

In this paper, we propose exact and fast L1 instruction/data cache configuration simulation algorithms for embedded applications that use PLRU or FIFO as a cache replacement policy. Firstly, we prove that the CRCB method can be applied not only to LRU but also to other cache replacement policies including FIFO and PLRU. Secondly, we prove several properties for FIFO- and PLRU-based caches and we propose associated cache simulation algorithms which can simulate simultaneously more than one cache configurations with different cache associativities accurately for FIFO or PLRU. Finally, many experimental results demonstrate that our cache configuration simulation algorithms obtains accurate cache hit/miss counts and runs up to 249 times faster than a conventional cache simulator.

This paper is organized as follows: Section 2 introduces the two cache configuration simulation methods: the Janapsatya method and the CRCB method, both of which assume LRU as a cache replacement policy. Furthermore, we prove that the Janapsatya method cannot be applied to FIFO nor PLRU; Section 3 explains several properties in the CRCB method and proves that the CRCB method can be applied to cache replacement policies including FIFO and PLRU; Section 4 proves several properties related to FIFO and proposes its associated fast and exact cache configuration simulation algorithm. Section 5 proves several properties related to PLRUt, one of the most typical PLRU cache replacement policies, and proposes its associated fast and exact cache configuration simulation algorithm. Section 6 gives concluding remarks and future works.

2. Fast Cache Configuration Simulation

Cache configuration is parameterized by the number of sets, block size, and associativity. We assume one of the three cache replacement policies, *LRU*, *FIFO*, and *PLRU*. Each *set* in a cache is represented by a priority queue and the number of sets shows the number of these priority queues. The *block size* is the smallest unit of data. *Associativity* shows the number of elements in each priority queue. Each element in a priority queue contains a cache tag. Let $c = (s, b, a)$ be a cache configuration where the number of sets is s, block size is b, and associativity is a. Then the tag, index, and offset for a memory address A is given as in **Fig. 1**. Let $S(c, i)$ and $S(c, i)_j$ be the i-th set or priority queue on c and the element whose priority is j in $S(c, i)$, where $j = 0, \cdots, (a-1)$. For example, if $c = (16, 16, 4)$ and $A = \underline{1010\,1010}\,\underline{\underline{0000}}\,0000$ are given, its tag becomes $\underline{1010\,1010}$ and its index becomes $\underline{\underline{0000}}$. The set indicated by $S(c, 0000)$ and the elements indicated by $S(c, 0000)_3$ are shown in **Fig. 2**. In this case, A is hit at $S(c, 0000)_3$.

Let s_0 and b_0 be the minimum number of sets and the minimum block size, respectively. Let s_m, b_m, and a_m be the maximum number of sets, the maximum block size, and the maximum associativity, respectively. Then we explore cache configurations (s, b, a) where

$$s = s_0, 2s_0, 4s_0, \ldots, s_m \tag{1}$$

$$b = b_0, 2b_0, 4b_0, \ldots, b_m \tag{2}$$

$$a = 1, 2, 3, \ldots, a_m \tag{3}$$

A naive exhaustive algorithm for cache simulation for a particular cache configuration $c = (s, b, a)$ and a particular memory address A is shown as follows:

[Exhaustive cache configuration simulation algorithm]

1) The memory address A is partitioned into its tag t, index i, and offset o. Calculate

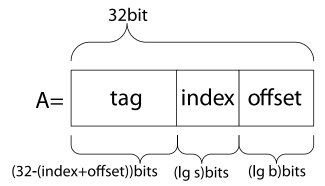


Fig. 1 Tag, index and offset.

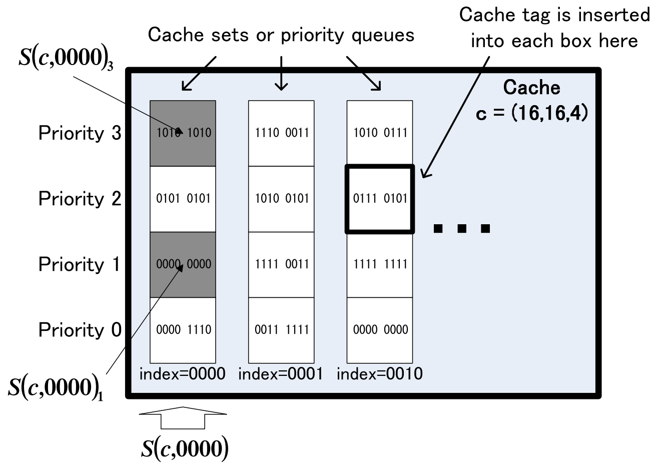


Fig. 2 $S(c, 0000)$, $S(c, 0000)_1$ and $S(c, 0000)_3$.

$S(c, i)$ by using c and i.

2) Check whether the priority queue $S(c, i)$ contains the tag t or not.

3) If $S(c, i)$ contains t at Step 2), a cache hit occurs on c for A and update $S(c, i)$ according to the cache replacement policy.

4) If $S(c, i)$ does not contain t on Step 2), a cache miss occurs on c for A. Add t into $S(c, i)$ and update it according to the cache replacement policy.

Step 1), Step 3) and Step 4) require $O(1)$ hit/miss *check counts*. Step 2) checks one by one a maximum of a elements and then Step 2) requires $O(a)$ hit/miss check counts. This means that a cache simulation for a memory address A on a cache configuration c requires $O(a)$ hit/miss check counts.

We repeat the above Step 1)–Step 4) for all the memory accesses and for all the possible cache configurations. Since the possible cache configurations are given by Eqs. (1)–(3), the number of cache configurations becomes $O(\lg s_m \times \lg b_m \times a_m)$. Let n be the number of total memory accesses. Then the exhaustive algorithm requires totally $O(n \times \lg s_m \times \lg b_m \times a_m^2)$ hit/miss check counts.

In 2006, Janapsatya, et al.[9] first proposed an extremely boosting up approach for the exhaustive algorithm. Their approach is based on LRU-based cache and runs up to 45 faster than the exhaustive-based approach which is called Dinero IV[3]. In 2009, Tojo, et al.[13] proposed the CRCB method for further

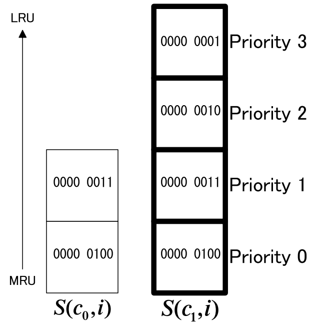


Fig. 3 The Janapsatya's method. It only checks the cache configuration with maximum associativity.

boosting up the Janapsatya's method. It is also based on LRU-based cache. Combined Janapsatya's method and CRCB method runs up to 3.3 times faster than Janapsatya's method only and 205 times faster than the exhaustive algorithm. As far as we know, this is the fastest approach for cache configuration simulation[⋆1].

In the rest of this section, we first prove that Janapsatya's method can run based on LRU but it cannot run based on other cache replacement policies such as FIFO and PLRU. After that, we review the CRCB method very briefly.

2.1 Janapsatya's Method

The Janapsatya's method[9] is a fast cache configuration simulation method which can be applied to LRU. It can simulate a_m cache configurations simultaneously by just simulating a single cache configuration $c = (s, b, a_m)$. In the LRU-based priority queue, we assume that the most-recent-used tag has the priority zero, the second most-recent-used tag has the priority one, and the least-recent-used tag has the priority $(a_m - 1)$.

Let us focus on the i-th set $S(c_0, i)$ on an LRU-based cache $c_0 = (s, b, 2)$ whose

⋆1 Recently, Haque, et al.[5)–7)] proposed the cache configuration simulation algorithms SuSeSim, DEW, and SCUD for LRU-based and FIFO-based L1 cache configuration simulation. However, overall cache configuration simulation including the three parameters of the number of sets, block size, and associativity is not discussed[5)–7)]. In that sense, we can say that combined Janapsatya's method and CRCB method is still the fastest one for LRU-based L1 cache simulation.

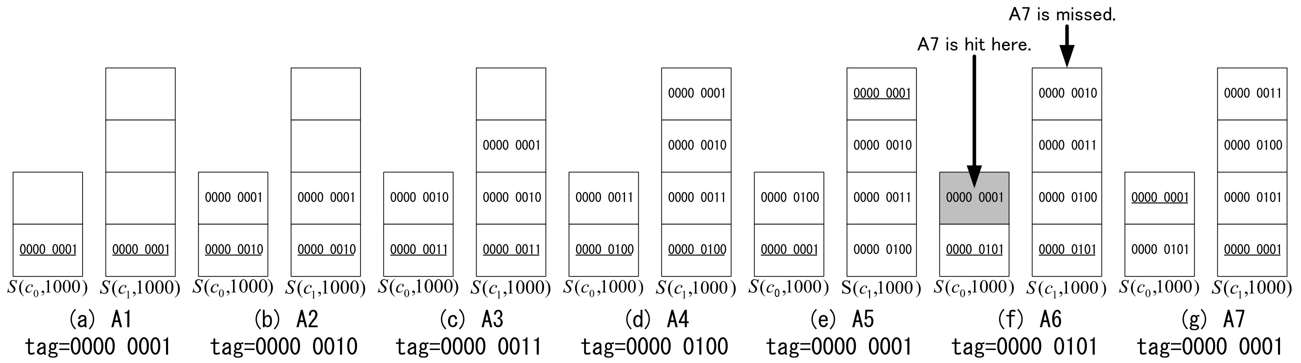


Fig. 4 Janapsatya's method cannot be applied to FIFO. In (f), the smaller cache set is no longer included in the larger cache set.

associativity is two. Let us also focus on the i-th set $S(c_1, i)$ on an LRU-based cache $c_1 = (s, b, 4)$ whose associativity is four. **Figure 3** shows the priority queues $S(c_0, i)$ and $S(c_1, i)$ and their associated priorities. As Fig. 3 shows, the tag 0000 0100 is used the most recently and 0000 0011 is used the second most recently. Assuming LRU as a cache replacement policy, if a cache hit occurs on c_0 at the tag of priority 0, then a cache hit also occurs on c_1 at the tag of priority 0. If a cache miss occurs on c_1, a cache miss also occurs on c_0. Even if a cache hit occurs on c_1 at the tag of priority 2 or priority 3, a cache miss occurs on c_0. If the number s of sets and block size b are fixed, we can accurately count cache hits/misses for every cache associativity (s, b, a) for $a = 1, 2, \ldots, a_m - 1$ by just simulating a cache configuration (s, b, a_m). This means that we can *skip* cache configuration simulations for (s, b, a) for $a = 1, 2, \ldots, a_m - 1$. This property is summarized as follows:

Property 1. *If a memory access causes a cache miss on a cache configuration (s, b, a), it also causes a cache miss on every cache configuration (s, b, a') where $a' = 1, 2, 3, \ldots, a - 1$. If a memory access causes a cache hit on a cache configuration (s, b, a), it also causes a cache hit on every cache configuration (s, b, a') where $a' = (a + 1), (a + 2), \ldots, a_m$.*

Janapsatya's method is based on Property 1. Janapsatya's method reduces the cache hit/miss check counts from $O(n \times \lg s_m \times \lg b_m \times a_m^2)$ to $O(n \times \lg s_m \times \lg b_m \times a_m)$. LRU-based caches satisfy Property 1 but we have the following theorem:

Theorem 1. *A FIFO-based cache does not satisfy Property 1.*

Proof. We show the counterexample. Let us consider a memory access sequence of Ai $(i = 1, \ldots, 7)$ below on FIFO-based cache configurations $c_0 = (16, 16, 2)$ and $c_1 = (16, 16, 4)$.

(1) A1 = 0000 0001 1000 0000
(2) A2 = 0000 0010 1000 1000
(3) A3 = 0000 0011 1000 0100
(4) A4 = 0000 0100 1000 0010
(5) A5 = 0000 0001 1000 0001
(6) A6 = 0000 0101 1000 0001
(7) A7 = 0000 0001 1000 0010

The cache set index of all these memory accesses Ai is the same and it is 1000. If a cache hit occur on c_0 or c_1, its tag must be stored in the set $S(c_0, 1000)$ or $S(c_1, 1000)$. **Figure 4** shows the contents of the sets $S(c_0, 1000)$ and $S(c_1, 1000)$ for these seven memory accesses. Just before the memory access A7 occurs (see Fig. 4 (f)), the set $S(c_0, 1000)$ contains 0000 0001 and 0000 0101 and the set $S(c_1, 1000)$ contains 00000010, 00000011, 00000100, and 00000101. If A7 is given, we can have a cache hit on c_0 but a cache miss on c_1 even though c_0 is smaller than c_1. A FIFO-based cache does not satisfy the Property 1 above. □

We can say similarly that a PLRUt-based cache, one of the PLRU-based caches, does not satisfy Property 1 either.

2.2 CRCB Method

The CRCB method[13] is also a fast cache configuration simulation method which can be applied to LRU. By using the general cache properties, the CRCB method skips several cache configuration simulations but obtains their cache hit/miss counts correctly. The CRCB method is composed of CRCB-1 and CRCB-2 summarized as follows:

CRCB-1: If a memory access causes a cache hit on a cache configuration $(s, b, 1)$, it also causes a cache hit on every cache configuration (s', b, a') where $s' = s, 2s, 4s, \ldots, s_m$ and $a' = 1, 2, 3, \ldots, a_m$. In this case, we can skip these cache configuration simulations (s', b, a').

CRCB-2: Let A_{i-1} and A_i be the two consecutive memory accesses. If their indexes are identical on a cache configuration (s, b, a) and also their tags are identical on (s, b, a), a cache hit always occurs on (s', b', a') for A_i where $s' = s, 2s, 4s, \ldots, s_m$, $b' = b, 2b, 4b, \ldots, b_m$, and $a' = 1, 2, \ldots, a_m$.

If we use both Janapsatya's method and CRCB method, cache hit/miss check counts are reduced to $O(n \times S \times B \times a_m)$ where $S \leq \lg s_m$ and $B \leq \lg b_m$. $\lg s_m$ and $\lg b_m$ can be reduced to S and B by reducing several cache configurations by the CRCB method.

As we have mentioned earlier, the CRCB method realizes the world fastest cache simulation. Then we try to apply it to FIFO- or PLRU-based cache simulation. In the next section, we prove that the CRCB method can be applied not only to LRU but also to FIFO and PLRUt.

3. Properties in the CRCB Method and the Cache Replacement Policies to which the CRCB Method can be Applied

The CRCB method uses the following two properties assuming LRU as a cache replacement policy:

Property 2. *If a memory access causes a cache hit on a cache configuration $(s, b, 1)$, it also causes a cache hit on every cache configuration (s', b, a') where $s' = s, 2s, 4s, \ldots, s_m$ and $a' = 1, 2, 3, \ldots, a_m$.*

Property 3. *Let A_{i-1} and A_i be the two consecutive memory accesses. If their indexes are identical on a cache configuration (s, b, a) and also their tags are identical on (s, b, a), a cache hit always occurs on (s', b', a') for A_i where $s' = s, 2s, 4s, \ldots, s_m$, $b' = b, 2b, 4b, \ldots, b_m$, and $a' = 1, 2, \ldots, a_m$.*

CRCB-1 uses Property 2 and CRCB-2 uses Property 3. Property 2 is further partitioned into the following two sub-properties.

Sub-property 2-1: If a memory access causes a cache hit on a cache configuration $(s, b, 1)$, it also causes a cache hit on every cache configuration $(s', b, 1)$ where $s' = s, 2s, 4s, \ldots, s_m$.

Sub-property 2-2: If a memory access causes a cache hit on a cache configuration $(s, b, 1)$, it also causes a cache hit on every cache configuration (s, b, a') where $a' = 1, 2, 3, \ldots, a_m$.

Property 3 is further partitioned into the following two sub-properties.

Sub-property 3-1: Let A_{i-1} and A_i be the two consecutive memory accesses. If their indexes are identical on a cache configuration (s, b, a) and also their tags are identical on (s, b, a), their indexes are identical on every cache configuration (s', b', a') and also their tags are identical on (s', b', a') where $s' = s, 2s, 4s, \ldots, s_m$, $b' = b, 2b, 4b, \ldots, b_m$, and $a' = 1, 2, \ldots, a_m$.

Sub-property 3-2: Let A_{i-1} and A_i be the two consecutive memory accesses. If their indexes are identical on a cache configuration (s, b, a) and also their tags are identical on (s, b, a), a cache hit always occur on (s, b, a) for A_i.

Tojo, et al.[13] proves Property 2, Property 3, Sub-property 2-1, Sub-property 2-2, Sub-property 3-1 and Sub-property 3-2 assuming LRU as a cache replacement policy.

Now let us focus on a very simple cache replacement policy which just satisfies Property 4 below:

Property 4. *Let A_j and A_k be any two memory accesses to a cache $c = (s, b, a)$ such that:*

i) *the indexes of A_j and A_k are identical on c, i.e., the tags of these two memory addresses are inserted into the identical cache set $S(c, i)$ in c,*
ii) *the memory access A_k occurs after the memory access A_j occurs, and*
iii) *there are no cache accesses to $S(c, i)$ between A_j and A_k.*

Then, the following two points are satisfied:

a) *After the memory access of A_j occurs, the tag of A_j remains in $S(c, i)$ until the memory access A_k occurs.*
b) *If the tags of A_j and A_k are identical, we have a cache hit for A_k when the memory access A_k occurs and we do not need to update the priority queue representing*

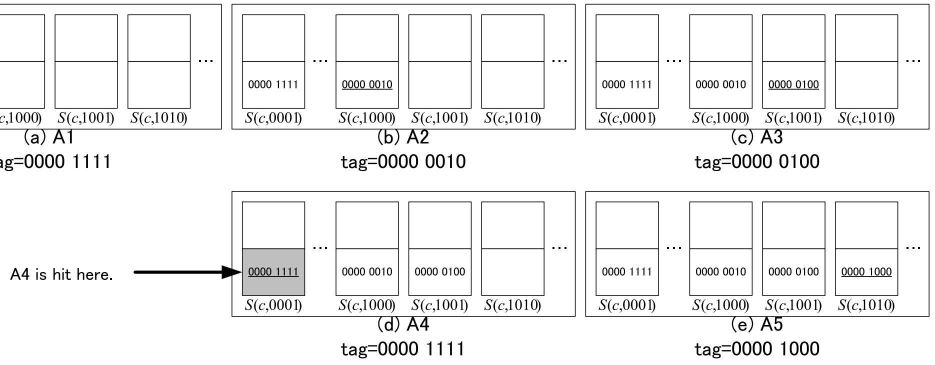


Fig. 5 An example of FIFO-based cache behavior when Property 4 is satisfied.

$S(c,i)$ at this time.

Assume that we have a memory access sequence of Ai ($i = 1, \dots, 5$) below assuming a cache configuration $(16, 16, 2)$:

(1) **A1 = 0000 1111 0001 0000**
(2) A2 = 0000 0010 1000 0000
(3) A3 = 0000 0100 1001 0000
(4) **A4 = 0000 1111 0001 0000**
(5) A5 = 0000 1000 1010 0000

Both of A1 and A4 have cache set index of 0001. A2 has cache set index of 1000 and A3 has cache set index of 1001. Property 4 focuses on the two memory accesses A1 and A4 whose cache set index is the same but there are no cache accesses to the cache set indexed by 0001 between them. For these two cache accesses, Property 4 claims that:

a) After the memory access of A1 occurs, the tag of A1, 0000 1111, remains in the cache set indexed by 0001 until the memory access A4 occurs.
b) If the memory access A4 occurs, we have a cache hit for A4 and we do not need to update the cache set indexed by 0001 at this time.

In this case, Property 4 also claims that a cache replacement policy satisfying Property 4 does not disturb nor update the cache set indexed by 0001 while the memory accesses A2 and A3 occur.

Assume that we consider FIFO as a cache replacement policy and the above memory access sequence of Ai ($i = 1, \dots, 5$) are given. **Figure 5** shows the cache behavior in this case. In FIFO-based cache, the tag of A1 remains in the cache set indexed by 0001 until the memory access A4 occurs and no update is required there. The memory accesses A2 and A3 do not disturb the cache set indexed by 0001.

Now we focus on a cache replacement policy satisfying Property 4[⋆1]. Property 4 is a very natural cache behavior but it satisfies the following very strong theorem:

Theorem 2. *If a cache replacement policy satisfies Property 4, the cache replacement policy also satisfies Property 2 and Property 3.*

First we prove that a cache replacement policy which satisfies Property 4 also satisfies Sub-property 2-1, Sub-property 2-2, Sub-property 3-1 and Sub-property 3-2.

Proof. (Sub-property 2-1) We focus on a cache configuration $c = (s, b, 1)$. Let A_j and A_k be two memory accesses, whose indexes are identical on c. The tags of these two memory addresses are inserted into the identical cache set $S(c, i)$. We assume that the memory access A_k occurs after the memory accesses A_j occurs and that there are no cache accesses to $S(c, i)$ between A_j and A_k.

⋆1 There may be cache replacement policies which do not satisfy Property 4. For example, the tag of A1 may not remain in the cache set indexed by 0001, or the memory access of A2 and/or A3 may disturb the cache set indexed by 0001 in the above example there. We only focus on here a cache replacement policy satisfying Property 4.

We further assume that the memory access A_k causes a cache hit in $S(c,i)$. According to Property 4, the tag of A_j remains in $S(c,i)$ until the memory access A_k occurs. Since the associativity of the cache c is one, $S(c,i)$ has a single tag, which must be the tag of A_j. Thus the tag of A_k must be the same as the tag of A_j.

Since the tags of A_j and A_k are identical and indexes of A_j and A_k are identical on $(s,b,1)$, the tags of A_j and A_k are identical and indexes of A_j and A_k are identical on $(s',b,1)$ where $s'=s,2s,\ldots,s_m$. The tags of these two memory addresses are inserted into the identical cache set $S(c',i')$ on $c'=(s',b,1)$. Since there are no cache accesses to $S(c',i')$ between A_j and A_k, we also have a cache hit for A_k in $S(c',i')$ according to Property 4.

In summary, if a memory access causes a cache hit on a cache configuration $(s,b,1)$, it also causes a cache hit on every cache configuration $(s',b,1)$ where $s'=s,2s,4s,\ldots,s_m$. A cache replacement policy satisfying Property 4 also satisfies Sub-property 2-1. □

Proof. (Sub-property 2-2) We focus on a cache configuration $c=(s,b,1)$. Let A_k and A_j be two memory accesses, whose indexes are identical on c, as in the proof of Sub-property 2-1. Then, the tag of A_k must be the same as the tag of A_j. Since the tags of A_j and A_k are identical and indexes of A_j and A_k are identical on $(s,b,1)$, the tags of A_j and A_k are identical and indexes of A_j and A_k are identical on (s,b,a') where $a'=1,2,\ldots,a_m$. The tag of these two memory addresses are inserted into the identical cache set $S(c',i')$ on $c'=(s,b,a')$. Since there are no cache accesses to $S(c',i')$ between A_j and A_k, we have a cache hit for A_k in $S(c',i')$ according to Property 4.

In summary, a cache replacement policy satisfying Property 4 also satisfies Sub-property 2-2. □

Proof. (Sub-property 3-1) Let A_{i-1} and A_i be the two consecutive memory accesses with l bits. If their indexes are identical on a cache configuration (s,b,a) and also their tags are identical on (s,b,a), $(l-\lg b)$ bits from the MSB of A_{i-1} and A_i are identical. Then $(l-\lg b')$ bits from the MSB of A_{i-1} and A_i are also identical on (s',b',a) where $s'=s,\ldots,s_m$ and $b'=b,\ldots,b_m$. This means that the tags of A_{i-1} and A_i are identical and indexes of A_{i-1} and A_i are also identical on (s',b',a') where $s'=s,\ldots,s_m$, $b'=b,\ldots,b_m$, and $a'=1,2,\cdots,a_m$. This is also true if we assume any cache replacement policy satisfying Property 4. We can say that a cache replacement policy satisfying Property 4 also satisfies Sub-property 3-1. □

Proof. (Sub-property 3-2) Let A_{i-1} and A_i be the two consecutive memory accesses. If their indexes are identical on any cache configuration (s,b,a) and also their tags are identical on (s,b,a), we can clearly say that a cache hit always occurs on (s,b,a) for A_i if we assume any cache replacement policy satisfying Property 4. We can also say that a cache replacement policy satisfying Property 4 also satisfies Sub-property 3-2. □

Since it is quite clear that Sub-property 2-1 and Sub-property 2-2 lead to Property 2 and Sub-property 3-1 and Sub-property 3-2 lead to Property 3, all of the above proofs give us the proof of Theorem 2. This means that we can use the CRCB method assuming any cache replacement policy satisfying Property 4.

Since Property 4 is a very natural property for a cache, not only LRU but also FIFO and PLRU satisfy Property 4. We can use the CRCB method even when we assume FIFO or PLRU as a cache replacement policy. If we use the CRCB method assuming FIFO or PLRU, its cache hit/miss check counts reduces to $O(n\times S\times B\times a_m^2)$, where $S\leq \lg s_m$ and $B\leq \lg b_m$. We expect that we can simulate cache configurations very fast even if they use FIFO or PLRU.

Now let us discuss how the maximum number s_m of sets, maximum block size b_m, and maximum associativity a_m affect the cache simulation speed in the CRCB method, assuming the minimum number s_0 of sets, minimum block size b_0, and minimum associativity of one are fixed.

CRCB-1: Since the CRCB-1 method is independent of block size, b_m cannot affect the cache simulation speed in CRCB-1. On the other hand, CRCB-1 can reduce more cache configuration simulations if s_m and a_m become larger as in Property 2.

CRCB-2: As Property 3 indicates, CRCB-2 can reduce more cache configuration simulations if s_m, b_m, and a_m become larger.

Overall the cache simulation speed using the CRCB method can be affected by s_m, b_m, and a_m but how much the simulation speed will be increased is heavily dependent on the simulation environment such as memory access histories and simulation computers.

The Janapsatya's method groups cache sets with different associativities and execute cache simulation simultaneously for them. On the other hand, the CRCB method itself does not group cache sets and thus cannot execute cache simulation simultaneously for them. If we can group several cache sets in FIFO- or PLRU-based cache, we can boost up cache configuration simulation furthermore. In Section 4 and Section 5, we propose a boosting up cache configuration simulation strategy by grouping several cache sets, assuming FIFO and PLRU, respectively.

4. Boosting up Cache Configuration Simulation based on FIFO as a Cache Replacement Policy

A cache replacement policy determines the behavior of the priority queue representing each set in a cache. Let a be an associativity of a cache. Then each priority queue representing a cache set contains at most a elements. FIFO means first-in-first-out and data in each priority queue are replaced in a first-in-first-out fashion. **Figure 6** shows an example of a priority queue when we have memory access sequences of **D**, **C**, **B**, and **A**. When **D** is inserted into a cache, **D** is the most IN-side. As **C**, **B**, and **A** are inserted into the priority queue, **D** is moved upward one by one to the OUT-side.

In this section, we propose a boosting up algorithm for cache configuration simulation assuming FIFO as a cache replacement policy.

4.1 Skipping the Priority Queue Update When its Behavior is the Same

Now let us focus on two cache configurations $c_0 = (s, b, a_0)$ and $c_1 = (s, b, a_1)$, where the number of sets s is identical, their block size b is also identical, but their associativity is different ($a_0 < a_1$). $S(c_0, i)$ shows a priority queue on c_0 whose index is i. As discussed in Section 2, the tag which is the most IN-side in $S(c_0, i)$ has the *priority* zero. The tag which is the second most IN-side in $S(c_0, i)$ has the *priority* one. The tag which is the most OUT-side in $S(c_0, i)$ has the *priority* $(a_0 - 1)$. **Figure 7** shows an example of priorities. In this figure, **D** is inserted first. Then **C**, **B**, and **A** are inserted in this order.

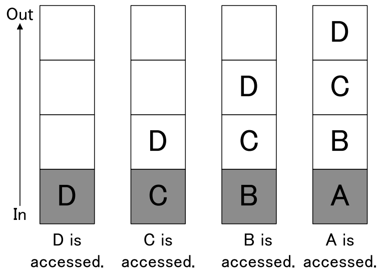


Fig. 6 The behavior of FIFO queue with associativity of four.

We assume here that all the tags in a priority queue $S(c_0, i)$ are included in a priority queue $S(c_1, i)$ and their priorities are also identical in both $S(c_0, i)$ and $S(c_1, i)$. In this case, we denote $c_0 \underset{i}{\subset} c_1$, i.e., c_0 is included in c_1 at index i. In other words, if $S(c_0, i)_j$ is equal to $S(c_1, i)_j$ for $0 \leq j \leq (a_0 - 1)$, we denote $c_0 \underset{i}{\subset} c_1$. Each of **Figs. 8** (a) and (b) shows the two cache sets for two cache configurations $c_0 = (s, b, 2)$ and $c_1 = (s, b, 4)$.

Figure 8 (a) shows the two cache sets $S(c_0, 0101)$ and $S(c_1, 0101)$ where $c_0 = (s, b, 2)$ and $c_1 = (s, b, 4)$. Figure 8 (b) shows the two cache sets $S(c_0, 1111)$ and $S(c_1, 1111)$ where $c_0 = (s, b, 2)$ and $c_1 = (s, b, 4)$. Figure 8 (a) focuses on the cache set index 0101. Figure 8 (b) focuses on the cache set index 1111. In case of Fig. 8 (a), we have $c_0 \underset{0101}{\subset} c_1$, but in case of Fig. 8 (b), we have $c_0 \underset{1111}{\not\subset} c_1$.

Let A be a memory access whose tag is *tag* and its index is i. Let $c_0 = (s, b, a_0)$ and $c_1 = (s, b, a_1)$ be two cache configurations assuming FIFO and $c_0 \underset{i}{\subset} c_1$ holds true for them. Then we have the following properties:

Property 5. *When tag is hit at $S(c_1, i)_j$ and $j < a_0$, we also have a cache hit on c_0 and $c_0 \underset{i}{\subset} c_1$ still holds. When tag is hit at $S(c_1, i)_j$ and $a_0 \leq j < a_1$, we have a cache miss on c_0 and we have $c_0 \underset{i}{\not\subset} c_1$.*

Proof. When *tag* is hit at $S(c_1, i)_j$ and $j < a_0$, it is also hit at $S(c_0, i)_j$ since $S(c_0, i)_j = S(c_1, i)_j$. When we have a cache hit on c_0 and c_1, we do not have to update its priority queues since we assume FIFO. Then $c_0 \underset{i}{\subset} c_1$ still holds true.

When *tag* is hit at $S(c_1, i)_j$ and $a_0 \leq j < a_1$, it is not at $S(c_0, i)_k$ for $0 \leq k < a_0$. We will have a cache miss on c_0. Since *tag* has to be inserted into c_0, we have priority

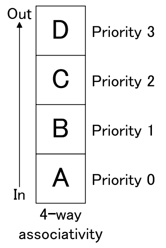


Fig. 7 Priorities in the FIFO queue.

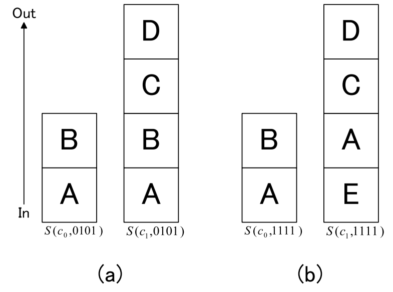


Fig. 8 (a) $c_0 \underset{0101}{\subset} c_1$ and (b) $c_0 \underset{1111}{\not\subset} c_1$.

queue update in c_0. Then we have $c_0 \underset{i}{\not\subset} c_1$. □

Property 6. *When we have a cache miss for tag on c_1, we also have a cache miss for tag on c_0 but $c_0 \underset{i}{\subset} c_1$ holds true.*

Proof. Since we have a cache miss for *tag* on c_1, we do not have *tag* at $S(c_1, i)_j$ where $0 \leq j < a_1$. Since $S(c_0, i)_j = S(c_1, i)_j$ for $j < a_0$, we do not have *tag* on c_0.

Since we have a cache miss on both c_0 and c_1, the priority queues indexed by i are updated and $c_0 \underset{i}{\subset} c_1$ still holds. □

Based on Properties 5 and 6, we can skip priority queue checking and updating in c_0 by just checking and updating priority queues in c_1 if $c_0 \underset{i}{\subset} c_1$ holds true. **Figure 9** shows how we can skip priority queue checking and updating. In this figure, we consider two cache configurations c_0 and c_1 and their two priority queues with index i, $S(c_0, i)$ and $S(c_1, i)$, where $c_0 \underset{i}{\subset} c_1$. $S(c_0, i)$ has two ways and $S(c_1, i)$ has four ways. When **B** is accessed, we only perform checking and updating of $S(c_1, i)$ and skip checking and updating of $S(c_0, i)$. In this case, $c_0 \underset{i}{\subset} c_1$ holds. Similarly, when **E** is accessed, we only perform checking and updating of $S(c_1, i)$ and skip checking and updating of $S(c_0, i)$. In this case, $c_0 \underset{i}{\subset} c_1$ holds. When **C** is accessed, we first check $S(c_1, i)$ and have a cache hit at the tag of priority two. Since we know that this tag is never included in $S(c_0, i)$, we have a cache miss on $S(c_0, i)$ without checking it. $S(c_0, i)$ must be updated so that **C** is inserted into it. After that, we have $c_0 \underset{i}{\not\subset} c_1$.

Note that $c_0 \underset{i}{\subset} c_1$ holds true if we have a cache hit on both c_0 and c_1 and we have a cache miss on both c_0 and c_1. Otherwise, we cannot guarantee $c_0 \underset{i}{\subset} c_1$.

We can also see that Properties 5 and 6 are a generalized version of Property 1. In other words, Property 1 is a very special case of ours which can be applied only to LRU-based cache.

4.2 The Algorithm

Based on Properties 5 and 6, we propose a boosting up algorithm for cache configuration simulation assuming FIFO as a cache replacement policy.

Let $C(s, b) = \{(s, b, a) \mid s$ and b are fixed and $a = 1, 2, \cdots, a_m\}$ be a set of cache configurations whose cache set size and block size are identical but their associativity is different. Let $SS(i) = \{S(c, i) \mid c \in C(s, b)\}$ be a set of cache sets whose index is i in $C(s, b)$. $SS(i)$ is partitioned into several *cache set groups* $SS(i)_1, SS(i)_2, \cdots$ as follows:

(1) For any cache configuration $c_0 \in C(s, b)$, there exists a group which contains $S(c_0, i)$.

(2) For any two cache configurations $c_0, c_1 \in C(s, b)$, if $c_0 \underset{i}{\subset} c_1$, $S(c_0, i)$ and $S(c_1, i)$ are in the same group.

Figure 10 shows an example of cache set groups. First, we assume that $a_m = 4$ and consider $C(s, b) = \{c_0 = (s, b, 1), c_1 = (s, b, 2), c_2 = (s, b, 3), c_3 = (s, b, 4)\}$. Then we have $SS(i) = \{S(c_0, i), S(c_1, i), S(c_2, i), S(c_3, i)\}$ as in Fig. 10. Since we

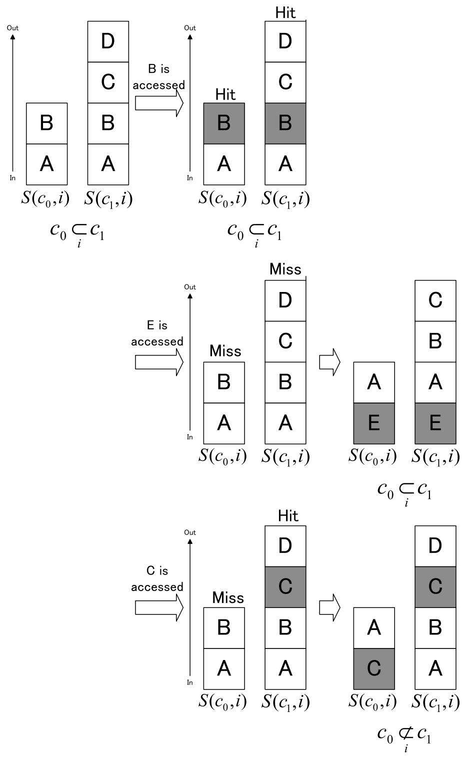


Fig. 9 Skipping cache configurations.

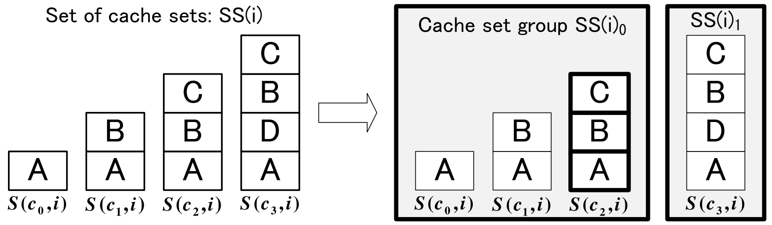


Fig. 10 Cache set groups.

have $c_0 \underset{i}{\subset} c_1$, $c_1 \underset{i}{\subset} c_2$, and $c_0 \underset{i}{\subset} c_2$, $S(c_0,i)$, $S(c_1,i)$ and $S(c_2,i)$ are included in the same group $SS(i)_0$. $SS(i)_1$ includes only $S(c_3,i)$. According to Properties 5 and 6, the priority queue $S(c_2,i)$ whose size is maximum can represent all the priority queue behaviors in $SS(i)_0$. This means that we only construct a data structure for the priority queue $S(c_2,i)$ in $SS(i)_0$.

Based on the above discussion, we first construct a single group $SS(i)_0$ in which all the cache sets with index i is included, since no tags are inserted into priority queues first and $c_0 \underset{i}{\subset} c_1$ is satisfied for any two cache sets $S(c_0,i)$ and $S(c_1,i)$. Then this single group is partitioned into several groups every time the condition (2) above is not satisfied. In each group, we only check and update the priority queue whose size is maximum.

We propose a cache-set-group based algorithm as follows:

[Cache-set-group based cache configuration simulation algorithm with CRCB]

```
1)  Generate a single cache set group including all the cache sets in $SS(i) = \{S(c,i) \mid c \in C(s,b)\}$ for each number of sets $s$, block size $b$, and cache set index $i$.
2)  For each memory access $A$, initialize the block size $b$ to be $b_0$, the number of sets $s$ to be $s_0$, and associativity $a$ to be two.
3)  If we can skip the block size $b$ by applying CRCB-2, go to Step 10).
4)  Based on CRCB-1, if we have a cache hit for $A$ on $(s,b,1)$, go to Step 9) since we can skip every cache configuration $(s',b,a')$ where $s' = s, 2s, ..., s_m$ and $a' = 1, 2, ..., a_m$.
5)  Let $S(c,i)$ in $c = (s,b,a)$ be a cache set that we have to check. Check cache hits/misses in the maximum-size priority queue $Q$ in the cache set group including $S(c,i)$.
    If cache hit occurs, go to Step 6).
    If cache miss occurs, update $Q$ and go to Step 7).
6)  If a cache hit occurs at priority $j$ in $Q$, we always have a cache hit in $(s,b,a')$ for $j < a'$ and skip cache hit/miss checks for them.
    In this case, we always have a cache miss in $(s,b,a')$ for $a' \le j$. Partition the cache set group and update their priority queues accordingly.
7)  Let $a$ to be the maximum size in the cache set group plus one.
    If the new $(s,b,a)$ is within the cache configuration range, go to Step 5).
8)  $a \leftarrow 2$ and $s \leftarrow 2s$.
    If the new $(s,b,a)$ is within the cache configuration range, go to Step 4).
9)  $s \leftarrow s_0$ and $b \leftarrow 2b$.
    If the new $(s,b,a)$ is within the cache configuration range, go to Step 3).
10) If there is a next memory access, go to Step 2). Otherwise, stop.
```

Our cache-set-group based algorithm itself reduces a cache hit/miss check counts from $O(n \times \lg s_m \times \lg b_m \times a_m^2)$ to $O(n \times \lg s_m \times \lg b_m \times a_m \times A)$ where $A \le a_m$. Combined CRCB method and our cache-set-group based algorithm reduces it to $O(n \times S \times B \times a_m \times A)$ where $S \le \lg s_m$, $B \le \lg b_m$ and $A \le a_m$.

Now let us discuss how the maximum number s_m of sets, maximum block size b_m, and maximum associativity a_m affect the cache simulation speed in our proposed cache-set-group based algorithm (excluding the CRCB method), assuming the minimum number s_0 of sets, minimum block size b_0, and minimum associativity of one are fixed.

Cache-set-group based algorithm: Since our cache-set-group based algorithm is independent of the number of sets and block size, s_m and b_m cannot affect the cache simulation speed in the CSG algorithm. On the other hand, the CSG algorithm can group more cache sets in a single group if a_m becomes larger as in Properties 5 and 6.

Overall the cache simulation speed using the CSG algorithm can be affected by a_m but how much the simulation speed will be increased is heavily dependent on the simulation environment such as memory access histories and simulation computers.

4.3 Experimental Evaluations

We have implemented our cache-set-group based algorithm in the C++ language on AMD 1.3 GHz CPU and 16 GB memory PC. We have obtained cache hit/miss counts for FIFO-based cache configurations by using Dinero IV [3], the CRCB method (CRCB), and combined CRCB method and our cache-set-group based algorithm (CRCB+CSG). Dinero IV is a cache simulator based on an exhaustive approach which checks cache hits/misses for a given cache configuration and memory access history. Dinero IV can be applied to LRU-based caches as well as FIFO-based caches. Note that, since we have proved that the CRCB method can be applied to FIFO-based caches in Section 3, it can be applied to

Table 1 Experimental comparisons 1 (FIFO-based cache configuration simulation).

	Dinero IV		CRCB		CRCB+CSG	
	CPU time [sec]	Hit/miss check counts [million]	CPU time [sec]	Hit/miss check counts [million]	CPU time [sec]	Hit/miss check counts [million]
ADPCM E	67 (1)	235 (1)	1.01 (0.015)	12 (0.051)	0.96 (0.014)	9 (0.038)
ADPCM D	68 (1)	228 (1)	1.04 (0.015)	12 (0.053)	0.98 (0.014)	10 (0.043)
ADPCM E (I)	669 (1)	4,796 (1)	4.83 (0.007)	58 (0.012)	4.30 (0.006)	38 (0.0079)
JPEG E	666 (1)	8,678 (1)	9.28 (0.014)	246 (0.028)	8.92 (0.013)	237 (0.027)
JPEG D	156 (1)	821 (1)	3.51 (0.023)	94 (0.11)	3.23 (0.021)	89 (0.11)
EPIC E	1,047 (1)	13,467 (1)	28.82 (0.028)	786 (0.058)	26.54 (0.025)	710 (0.053)
EPIC D	224 (1)	1,495 (1)	8.89 (0.040)	277 (0.19)	7.95 (0.035)	241 (0.16)
G721 E	5,833 (1)	66,465 (1)	29.80 (0.005)	423 (0.0064)	28.81 (0.005)	413 (0.0062)
G721 D	5,980 (1)	69,251 (1)	29.31 (0.005)	388 (0.0056)	28.64 (0.005)	379 (0.0055)

Table 2 Hit/miss counts (FIFO-based cache configuration simulation).

		Dinero IV		CRCB+CSG	
	Cache configuration	cache hit counts	cache miss counts	cache hit counts	cache miss counts
JPEG E	(32,8,1)	3,597,258	1,587,745	3,597,258	1,587,745
	(32,1024,8)	5,184,812	191	5,184,812	191
	(1024,256,16)	5,184,292	711	5,184,292	711
G721 E	(32,8,1)	39,293,727	8,608,067	39,293,727	8,608,067
	(32,1024,8)	47,901,763	31	47,901,763	31
	(1024,256,16)	47,901,686	108	47,901,686	108

FIFO-based cache configuration simulation.

Application programs that we have used here are from MediaBench[10] as in **Table 1**. In the table, E stands for encoder and D stands for decoder. We assume L1 instruction cache in ADPCM E (I) and we assume L1 data cache in other application programs. We have also used SimpleScalar[1] to obtain memory access histories. Simulated cache configurations are:

The number of sets: $32, 64, \ldots, 524{,}288$

Block size: $8, 16, \ldots, 1{,}024$bytes

Associativity: $1, 2, 4, \ldots, 16$.

We have totally 380 cache configurations whose total cache size ranges from 256 bytes to 4,194,304 bytes. Table 1 shows the experimental results. Obtained cache hit/miss counts for every cache configuration are exactly the same for Dinero IV, CRCB, and CRCB+CSG. **Table 2** shows the cache hit counts and cache miss counts for several cache configurations in JPEG E and G721 E. Roughly saying, CRCB+CSG runs up to 208 faster than Dinero IV. It finishes cache configuration simulation within several seconds while Dinero IV takes several hours. Our approach makes simulation-based cache hit/miss measuring a practical one.

Figures 11 and **12** show how many cache set groups are generated in JPEG E and G721 E as the memory accesses proceed. Theoretically, the minimum number of cache set groups is 2,088,704, in which each cache set group includes all the cache sets in $SS(i)$. The maximum number of cache set groups is 4,046,080, in which each cache set group includes only a single cache set. As shown in Figs. 11 and 12, the number of cache set groups do not reach its maximum value in all the simulation time. This is because a particular cache access pattern which

Table 3 CPU time comparison when $s_{\max}$ becomes smaller (FIFO-based cache configuration simulation).

	Dinero IV	CRCB	CRCB+CSG
	CPU time [sec]	CPU time [sec]	CPU time [sec]
ADPCM E	34 (1)	0.51 (0.015)	0.45 (0.013)
ADPCM E (I)	350 (1)	4.34 (0.012)	3.95 (0.011)
JPEG D	82 (1)	2.57 (0.031)	2.38 (0.029)
G721 E	3,087 (1)	29.47 (0.010)	28.81 (0.009)

$s_{\max}$ is set to be 512.

	Dinero IV	CRCB	CRCB+CSG
	CPU time [sec]	CPU time [sec]	CPU time [sec]
ADPCM E	60 (1)	0.64 (0.011)	0.58 (0.010)
ADPCM E (I)	608 (1)	4.41 (0.007)	4.02 (0.007)
JPEG D	140 (1)	3.09 (0.022)	2.85 (0.020)
G721 E	5,305 (1)	28.85 (0.005)	28.82 (0.005)

$s_{\max}$ is set to be 16,384.

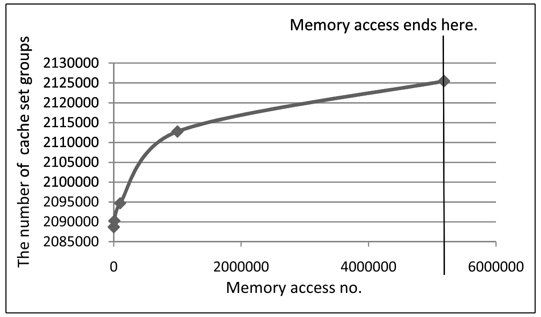


Fig. 11 Cache set groups on FIFO-based cache configuration simulation for JPEG E.

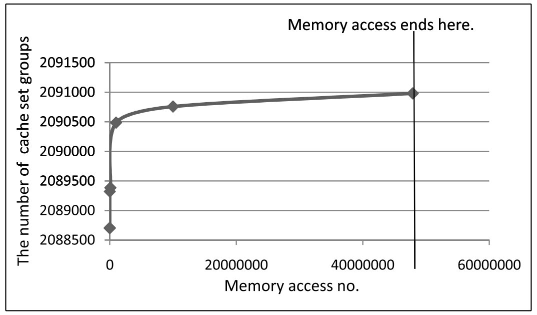


Fig. 12 Cache set groups on FIFO-based cache configuration simulation for G721 E.

partitions all cache set groups does not usually occur in practical memory access histories. This means that the overheads of making cache set groups can be too small compared with reducing cache configurations by using our CSG method.

The maximum number of sets, 524,288, can be too large for embedded processors. Then we have done additional experiments where the number of sets is limited to up to 16,384 or 512. **Table 3** summarizes these experimental results. Even in these experiments, CRCB+CSG runs 34 to 184 times faster than Dinero IV.

Note that SCUD[6)] was proposed very recently for FIFO-based L1 cache configuration simulation. It claims that SCUD runs up to 57 times faster than Dinero IV but it can be applied only when the cache block size and cache associativity are fixed. We cannot say that SCUD realizes cache configuration simulation including the overall three cache parameters of the number of sets, block size, and associativity. Since our proposed approach CRCB+CSG runs 28 to 208 times faster than Dinero IV for FIFO-based L1 cache as in Table 1, it means that our approach can be much faster than SCUD although we cannot compare them directly.

5. Boosting up a Cache Configuration Simulation based on PLRU as a Cache Replacement Policy

When implementing an original LRU-based cache into hardware, its costs can be too high because of its complex behavior. PLRU, or pseudo LRU, is a cache replacement policy which has cache hit rate as high as an original LRU but can be implemented very easily. PLRU does not show a particular algorithm but just shows LRU-like cache replacement policies. In this section, we pick up PLRUt[2)], one of the most typical PLRU cache replacement policies.

The PLRUt algorithm can be viewed as a complete binary tree, in which a cache tag is inserted into a *leaf* and each node other than leaves has one-bit *node*

Table 4 Experimental comparisons 2 (PLRUt-based cache configuration simulation).

	Exhaustive approach		CRCB		CRCB+CSG	
	CPU time [sec]	Hit/miss check counts [million]	CPU time [sec]	Hit/miss check counts [million]	CPU time [sec]	Hit/miss check counts [million]
ADPCM E	82 (1)	204 (1)	1.45 (0.022)	12 (0.059)	1.03 (0.017)	9 (0.044)
ADPCM D	81 (1)	204 (1)	1.51 (0.022)	13 (0.064)	1.07 (0.019)	10 (0.049)
ADPCM E (I)	988 (1)	4,710 (1)	5.77 (0.006)	60 (0.013)	4.87 (0.005)	41 (0.0087)
JPEG E	787 (1)	2,090 (1)	11.16 (0.017)	207 (0.099)	10.88 (0.016)	198 (0.095)
JPEG D	186 (1)	506 (1)	4.59 (0.029)	76 (0.15)	3.95 (0.025)	71 (0.14)
EPIC E	1,232 (1)	5,721 (1)	35.01 (0.033)	796 (0.14)	36.32 (0.034)	706 (0.12)
EPIC D	264 (1)	1,084 (1)	10.89 (0.049)	286 (0.26)	10.53 (0.047)	240 (0.22)
G721 E	7,073 (1)	35,481 (1)	30.92 (0.005)	299 (0.0084)	29.30 (0.005)	232 (0.0065)
G721 D	7,243 (1)	35,897 (1)	30.25 (0.005)	291 (0.0081)	29.04 (0.005)	239 (0.0067)

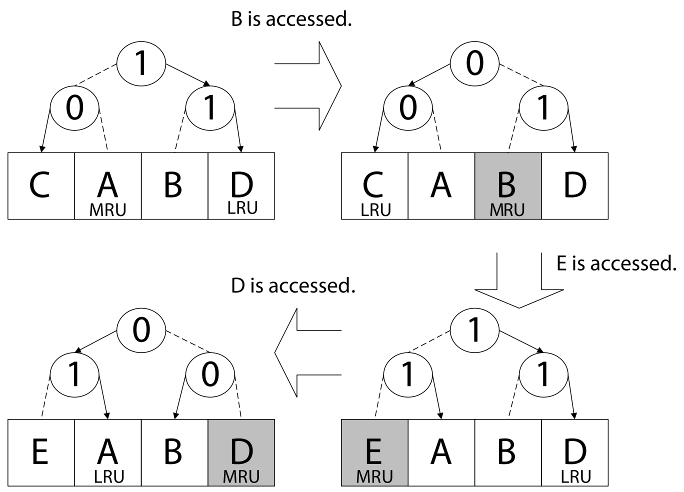


Fig. 13 PLRUt behavior.

flag. The one-bit node flag shows which child node is LRU or not. If it is zero, its left node is LRU. If it is one, its right node is LRU. If a memory address whose tag is t is accessed and a cache hit occurs, then the node flags from the root node to the leaf containing t are set to be one or zero, so that each of them does not point to t. If a cache miss occurs, we traverse from the root node according to the node flag. If the node flag is zero, we go to its left node and, if it is one, we go to its right node. We finally get to the leaf, which can be the LRU tag. We discard this tag, insert t into this leaf, and reverse every node flag from the root node to this leaf.

Figure 13 shows an example of PLRUt behavior. The arrow at each node shows its LRU child node. By traversing from the root, we can find out the LRU tag. Initially, the LRU tag is **D** and the MRU tag is **A**. If the memory access to **B** occurs, we update the arrows from the node to **B** so that they do not point to **B**. In this case, the LRU tag changes from **D** to **C**. After that, if the memory access to **E** which is not on the cache occurs, the LRU tag **C** is discarded and **E** is inserted here. We update all the arrows from the root node to **E** so that they do not point to **E**. After that, if **D** is accessed, we update the arrows similarly.

In this section, we propose a boosting up algorithm for cache configuration simulation assuming PLRUt as a cache replacement policy.

5.1 Skipping the Priority Queues' Update when their Behavior is the Same

First, we map a tree structure in PLRUt to a priority queue. Let t be a leaf in a tree structure in PLRUt. We define the priority $p(t)$ of the leaf t as follows: Consider the path from the root node to t which is composed of a sequence $\{v_0, v_1, \ldots, v_n\}$ of nodes, where n shows the level of the leaf t. In this case, the k-th bit of $p(t)$ is defined by one if the node flag of v_k points to v_{k+1} for $0 \leq k \leq (n-1)$. It is defined by zero if the node flag of v_k does not point to v_{k+1}. If all the leaves have their priority, we can consider them to be a priority

Table 5 Hit/miss counts (PLRUt-based cache configuration simulation).

		Exhaustive approach		CRCB+CSG	
	Cache configuration	cache hit counts	cache miss counts	cache hit counts	cache miss counts
JPEG E	(32,8,1)	3,597,258	1,587,745	3,597,258	1,587,745
	(32,1024,8)	5,184,801	202	5,184,801	202
	(1024,256,16)	5,184,292	711	5,184,292	711
G721 E	(32,8,1)	39,293,727	8,608,067	39,293,727	8,608,067
	(32,1024,8)	47,901,763	31	47,901,763	31
	(1024,256,16)	47,901,686	108	47,901,686	108

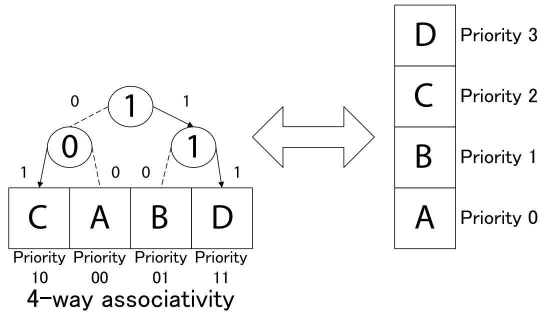


Fig. 14 Mapping a PLRUt tree to its priority queue.

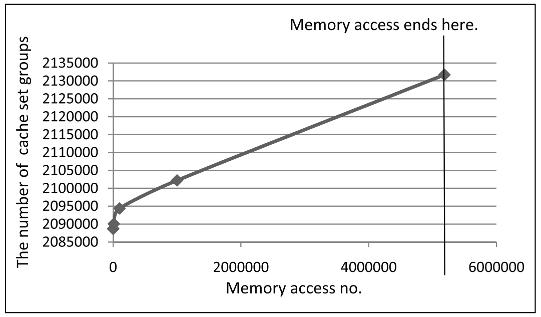


Fig. 15 Cache set groups on PLRUt-based cache configuration simulation for JPEG E.

queue. **Figure 14** shows an example of this mapping from the tree to the priority queue.

We can have the same discussions as Section 4 to these mapped priority queues. Since Properties 5 and 6 just discuss a general priority queue, they hold true for priority queues mapped from PLRUt trees. Based on Properties 5 and 6, we also define a *cache set group* as in the same discussion in the previous section.

Note that it is natural that the associativity a in PLRUt-based cache is $1, 2, 4, 8, \ldots, a_m$.

5.2 The Algorithm and Experimental Evaluations

By mapping a PLRUt tree into its priority queue, the cache-set-group based algorithm proposed in Section 4 can also simulate cache configurations for PLRUt-based caches.

As far as we know, there are no boosting up cache simulation algorithms for PLRUt-based caches. Then we have used the exhaustive approach discussed in Section 2, the CRCB method (CRCB), and combined CRCB method and our cache-set-group based algorithm (CRCB+CSG) and obtained cache hit/miss counts for PLRUt-based cache configurations. The experimental environments are the same as those in Section 4.

Table 4 shows the experimental results. Obtained cache hit/miss counts for every cache configuration are exactly the same for the exhaustive approach, CRCB, and CRCB+CSG. **Table 5** shows the cache hit counts and cache miss counts for several cache configurations in ADPCM E and G721 E. As Table 4 indicates, CRCB+CSG runs up to 249 times faster than the exhaustive approach here. It simply means that our approach CRCB+CSG achieves the world fastest cache configuration simulation for PLRUt-based L1 cache simulation.

Figures 15 and **16** show how many cache set groups are generated in JPEG

Table 6 CPU time comparison when $s_{\max}$ becomes smaller (PLRUt-based cache configuration simulation).

	Dinero IV	CRCB	CRCB+CSG
	CPU time [sec]	CPU time [sec]	CPU time [sec]
ADPCM E	42.63 (1)	0.55 (0.013)	0.53 (0.012)
ADPCM E (I)	517 (1)	4.80 (0.009)	4.51 (0.009)
JPEG D	98 (1)	3.11 (0.032)	3.09 (0.032)
G721 E	3,698.19 (1)	30.33 (0.008)	29.24 (0.008)

$s_{\max}$ is set to be 512.

	Dinero IV	CRCB	CRCB+CSG
	CPU time [sec]	CPU time [sec]	CPU time [sec]
ADPCM E	73.54 (1)	0.76 (0.010)	0.67 (0.009)
ADPCM E (I)	897 (1)	5.00 (0.006)	4.50 (0.005)
JPEG D	169 (1)	3.81 (0.023)	3.61 (0.021)
G721 E	6,419.71 (1)	30.06 (0.005)	29.15 (0.005)

$s_{\max}$ is set to be 16,384.

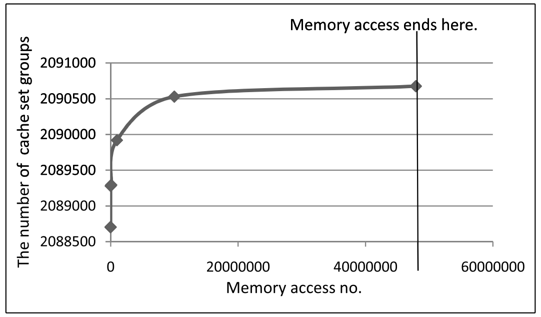


Fig. 16 Cache set groups on PLRUt-based cache configuration simulation for G721 E.

E and G721 E as the memory accesses proceed. Theoretically, the minimum number of cache set groups is 2,088,704 and the maximum number of cache set groups is 4,046,080. As shown in Figs. 15 and 16, the number of cache set groups do not reach its maximum value in all the simulation time in this case as well. Overheads of making cache set groups can be too small compared with reducing cache configurations by using our CSG algorithm.

As in the same discussion in the previous section, the maximum number of sets, 524,288, can be too large for embedded processors. Then we have done additional experiments where the number of sets is limited to up to 16,384 or 512. **Table 6** summarizes these experimental results. Even in these experiments, CRCB+CSG runs 31 to 220 times faster than the exhaustive approach.

6. Conclusions

In this paper, we proposed exact and fast L1 cache configuration simulation algorithms for embedded applications that use PLRU or FIFO as a cache replacement policy. Firstly, we proved that the CRCB method can be applied not only to LRU but also to other cache replacement policies including FIFO and PLRU. Secondly, we proved several properties for FIFO- and PLRU-based caches and we proposed associated cache simulation algorithms which can simulate simultaneously more than one cache configurations with different cache associativities accurately. Finally, experimental resulted demonstrate that our cache configuration simulation algorithms obtained accurate cache hit/miss counts and runs up to 249 times faster than a conventional cache simulator.

In the future, we will extend our cache configuration simulation to a hierarchical cache memory system including L1/L2 caches.

Acknowledgments The research is supported in part by KAKENHI (No. 22300019), Waseda University Grant for Special Research Projects (No. 2011A-045), the Telecommunication Advancement Foundation, and the Kurata Memorial Hitachi Science and Technology Foundation.

References

1) Austin, T., Larson, E. and Ernst, D.: SimpleScalar: An infrastructure for computer system modeling, *IEEE Trans. Comput.*, Vol.35, No.2, pp.59–67 (2002).
2) Berg, C.: PLRU cache domino effects, *Proc. 6th Int. Workshop WCET Analysis*, Mueller, F. (Ed), pp.29–31 (2006).
3) Edler, J. and Hill, M.D.: Dinero IV trace-driven uniprocessor cache simulator, available from ⟨http://www.cs.wisc.edu/~markhill/DineroIV/⟩.
4) Fornaciari, W., Sciuto, D., Silvano, C. and Zaccaria, V.: A design framework to efficiently explore energy-delay tradeoffs, *Proc. 9th International Symposium on Hardware/Software Codesign*, pp.260–265 (2001).
5) Haque, M.S., Janapsatya, A. and Parameswaran, S.: SuSeSim: A fast simula-

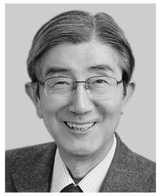
tion strategy to find optimal L1 cache configuration for embedded systems, *Proc. CODES+ISSS' 09*, pp.295–304, ACM (2009).
6) Haque, M.S., Peddersen, J., Janapsatya, A. and Parameswaran, S.: DEW: A fast level 1 cache simulation approach for embedded processors with FIFO replacement policy, *Proc. DATE 2010*, pp.496–501 (2010).
7) Haque, M.S., Peddersen, J., Janapsatya, A. and Parameswaran, S.: SCUD: A fast single-pass L1 cache simulation approach for embedded processors with round-robin replacement policy, *Proc. DAC 2010*, pp.356–361 (2010).
8) Hill, M.D. and Smith, A.J.: Evaluating associativity in CPU caches, *IEEE Trans. Comput.*, Vol.38, No.12, pp.1612–1630 (1989).
9) Janapsatya, A., Lgnjatovic, A. and Parameswaran, S.: Finding optimal L1 cache configuration for embedded systems, *Proc. ASP-DAC 2006*, pp.796–801 (2006).
10) Lee, C., Potkonjak, M. and Mangione-Smith, W.H.: MediaBench: A tool for evaluating and synthesizing multimedia and communications systems, *Proc. 30th Annual International Symposium on Microarchitecture* (1997).
11) Pieper, J.J., Mellan, A., Paul, J.M., Thomas, D.E. and Karim, F.: High level cache simulation for heterogeneous multiprocessors, *Proc. 41st Design Automation Conference*, pp.287–292 (2004).
12) Sugumar, R.A.: Set-associative cache simulation using generalized binomial trees, *ACM Trans. Comput. Syst.*, Vol.13, No.1, pp.32–56 (1995).
13) Tojo, N., Togawa, N., Yanagisawa, M. and Ohtsuki, T.: Exact and fast L1 cache simulation for embedded systems, *Proc. ASP-DAC 2009*, pp.817–822 (2009).

(Received November 28, 2010)
(Revised March 3, 2011)
(Accepted April 24, 2011)
(Released August 10, 2011)

(Recommended by Associate Editor: *Tsuyoshi Isshiki*)

Masashi Tawada received his B. Eng. degree from Waseda University in 2010 in computer science. He is presently working toward M. Eng. degree there. His research interests are cache design and embedded architecture.

Masao Yanagisawa received his B. Eng., M. Eng. and Dr. Eng. degrees from Waseda University in 1981, 1983, and 1986, respectively, all in electrical engineering. He was with University of California, Berkeley from 1986 through 1987. In 1987, he joined Takushoku University. In 1991, he left Takushoku University and joined Waseda University, where he is presently a Professor in the Department of Computer Science and Engineering. His research interests are combinatorics and graph theory, computational geometry, VLSI design and verification, and network analysis and design. He is a member of IEEE, ACM, and IEICE.


Tatsuo Ohtsuki received his B. Eng., M. Eng., Dr. Eng. degrees from Waseda University in 1963, 1965, and 1970, respectively, all in electrical engineering. In 1965, he joined the NEC Corporation Ltd., Tokyo, Japan. From 1978 to 1980, he served as Research Manager, Application System Research Laboratory, at Central Research Laboratories. In 1980, he left NEC and joined Waseda University, where he is presently a Professor in the Department of Computer Science and Engineering. His research interests are algorithm and hardware engines for VLSI design and verification, computer algorithms for combinatorial problems, and network analysis/design. He is a fellow of IEEE, and a fellow of IEICE.

Nozomu Togawa received his B. Eng., M. Eng. and Dr. Eng. degrees from Waseda University in 1992, 1994, and 1997, respectively, all in electrical engineering. He is presently a Professor in the Department of Computer Science and Engineering, Waseda University. His research interests are VLSI design, graph theory, and computational geometry. He is a member of IEEE and IEICE.