Dueling CLOCK: Adaptive Cache Replacement Policy Based on The CLOCK Algorithm

Andhi Janapsatya, Aleksandar Ignjatović†, Jorgen Peddersen and Sri Parameswaran†
School of Computer Science & Engineering, University of New South Wales, Sydney, NSW 2052, Australia
†NICTA, Sydney, NSW 2052, Australia
{andhij, ignjat, jorgenp, sridevan}@cse.unsw.edu.au

ABSTRACT

We consider the problem of on-chip L2 cache management and replacement policies. We propose a new adaptive cache replacement policy, called Dueling CLOCK (DC), that has several advantages over the Least Recently Used (LRU) cache replacement policy.

LRU's strength is that it keeps track of the 'recency' information of memory accesses. However, a) LRU has a high overhead cost of moving cache blocks into the most recently used position each time a cache block is accessed; b) LRU does not exploit 'frequency' information of memory accesses; and, c) LRU is prone to cache pollution when a sequence of single-use memory accesses that are larger than the cache size is fetched from memory (i.e., it is non scan resistant).

The DC policy was developed to have low overhead cost, to capture 'recency' information in memory accesses, to exploit the 'frequency' pattern of memory accesses and to be scan resistant. In this paper, we propose a hardware implementation of the CLOCK algorithm for use within an on-chip cache controller to ensure low overhead cost. We then present the DC policy, which is an adaptive replacement policy that alternates between the CLOCK algorithm and the scan resistant version of the CLOCK algorithm.

We present experimental results showing the MPKI (Misses per thousand instructions) comparison of DC against existing replacement policies, such as LRU. The results for an 8-way 1MB L2 cache show that DC can lower the MPKI of SPEC CPU2000 benchmark by an average of 10.6% when compared to the tree based Pseudo-LRU cache replacement policy.

1. Introduction

Caching techniques are fundamental for bridging the performance gap between components in a computer system, such as the performance of processor and memory. Performance of caching techniques has great influence over memory latency, processor performance and energy consumption. Cache replacement policies for on-chip caches need to have good miss rate performance as well as low maintenance cost. Low maintenance cost is imperative due to the operational frequency of the on-chip caches. Thus, it is not feasible to implement a cache replacement policy that has a large maintenance overhead.

Consider a system with three memory levels. At the highest level, the processor is connected to two separate L1 caches (one for instruction cache and the other for data cache). The two L1 caches are then connected to a unified L2 cache and the L2 cache is then connected to an off-chip main memory. Figure 1 depicts such a system with three levels of memory hierarchy.

Traditionally, the Least Recently Used (LRU) cache replacement policy has been considered to be one of the best policies to be used for on-chip cache replacement. However, LRU has many disadvantages as stated by Bansal and Modha in [1]. These disadvantages are:

- High maintenance cost. LRU needs to move a cache block into the most recently used (MRU) position on every cache hit.
- LRU only captures the 'recency' features of memory accesses and it does not exploit the 'frequency' features of memory accesses.
- The LRU stack can easily be polluted by a scan, which is a sequence of single-use memory accesses.

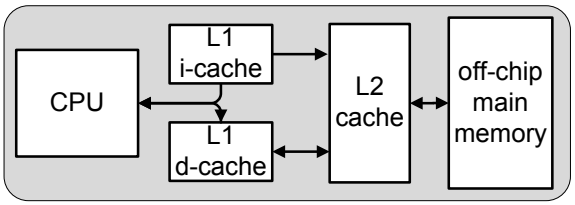

Figure 1: System Architecture

The high maintenance cost of LRU has led to many approximation of the LRU policy for use in on-chip caches. One such implementation is the tree-based Pseudo-LRU (PLRUt) policy.

In this paper, we propose a new cache replacement policy that is targeted towards the L2 cache. We call our cache replacement policy, Dueling CLOCK (DC). The development of the cache replacement policy DC was inspired by the CLOCK algorithm originally developed for the MULTICS system in the 1960s. The features of DC are its low maintenance cost, capturing the 'recency' information in memory accesses, exploitation of the 'frequency' features of memory accesses and scan resistance.

The remainder of this paper is structured as follows. We present a summary of existing cache replacement policies and the reasoning behind the development of DC in Section 2; Section 3 will describe the DC algorithm; Section 4 describes the experimental setup and the procedure to generate the memory trace benchmarks; Section 5 presents the results of performance comparison; and finally Section 6 concludes this paper.

2. Related Work

The replacement policy for a cache is typically developed according to its operational need. In this section, we present a brief survey of on-chip cache replacement policies that are targeted for L2 caches. We will also present the motivation behind the development of the DC policy.

LRU uses a stack to record 'recency' information of the memory accesses. The top of the stack indicates MRU cache block and the bottom of the stack indicates LRU cache block. Upon every cache hit, the hit cache block will be moved to the MRU position. For a 2-way associative cache, the implementation of LRU is trivial and its implementation cost is minimal. However, as associativity gets larger than

978-3-9810801-6-2/DATE10 © 2010 EDAA

4, no feasible implementation is available. The simplest implementation, as a hardware stack of cache addresses, requires $N \times log_2(N)$[1] bits to store the necessary information, and the large delays necessary for moving elements to the top of the stack as they are accessed are infeasible for on-chip implementation. Grossman [5] provides an implementation utilizing a systolic array to replace the stack to reduce the logic complexity. However, this is performed by doubling the number of utilized storage bits. It should be noted that the number of storage elements or the logic required grows exponentially as the number of associativity grows.

Loper et al. [10] presented a modification to limit the number of bits required to encode all the LRU order permutations. Their method requires only 5 bits to represent a 4-way cache, though the logic complexity for larger associativity is infeasible and was not analyzed by the authors. Handy [6] suggests an implementation that encodes the possible permutations of the LRU order and utilizes state transitions to move between them. This reduces the amount of storage space required to $log_2(N!)$, but requires extremely complex logic to implement (e.g., an 8-way LRU implementation would need 40,320 states in a 16-bit encoding with 8 transitions per state).

The PLRUt replacement policy minimizes the maintenance cost of updating hit cache block into the MRU position while attempting to approximate LRU behavior. Commercial desktop processors, such as the Intel Pentium 4, use the PLRUt [9]. PLRUt uses a binary tree to approximate the recency stack in LRU [6]. For example, given a 4-way associative cache, three bits (in a two level binary tree) can be used to keep track of the recency stack. The top bit in the tree indicates which half of the cache was least recently hit. The second level of the tree is used to indicate which cache block in each half is the least recently used of that half, The pseudo least recently used cache block will thus be the least recently used cache block of the least recently used half. Each time the cache is accessed, PLRUt needs to update each level of the tree such that the hit cache-way (or the LRU cache-way in the case of a cache miss) is the MRU cache block of the tree. The extra memory required for PLRUt grows linearly with associativity.

A cache replacement policy introduced by Robinson and Devarakonda uses access frequency to determine which cache block should be evicted upon a cache miss. The policy is called frequency based replacement (FRB) [14]. FRB counts the number of accesses to each cache block, and it selects the cache block with the lowest access frequency as the victim cache block for replacement. The authors of [14] showed that FRB has lower miss rates compared to LRU. However, FRB requires multiple counters to count access frequency of each cache block in the cache which makes implementation infeasible and the operational cost prohibitively expensive.

The FIFO (also known as round-robin) replacement policy uses a *replacePtr* to indicate the cache block that is to be replaced when a cache miss occurs. FIFO does not record 'recency' information nor does it exploit the 'frequency' of memory accesses and it is known to have lower performance than LRU. Second Chance (SC) is an improvement from FIFO. SC uses a reference bit for each cache block. The reference bit will be set to 'one' each time the cache block is accessed (i.e., a cache hit occurs on the cache block). SC uses a queue, where the head of the queue represents the next cache block to be replaced upon a cache miss. The deficiency of SC is the need to keep moving cache blocks from the head of the queue to the tail.

The CLOCK algorithm has identical functionality to SC with a more efficient implementation by using a circular queue. CLOCK uses a circular queue and a *replacePtr* to indicate the next cache block to be replaced upon a cache miss. The use of the circular queue avoids the movement of cache blocks from the head of the queue to the tail of the queue, instead it replaces the operation by advancing the *replacePtr* to point to the next cache block in the circular queue. It should be noted that the CLOCK algorithm was developed for low-overhead and low-lock-contention environment for use as a page replacement policy [1]. To the best of our knowledge, no feasible hardware implementation for on-chip caches using CLOCK algorithm has been previously published.

All the cache replacement policies that have been presented above are static replacement policies. In order to improve performance, it has been identified that the application's memory access behavior may change during the course of execution and different applications exhibit different memory access patterns.

Several cache replacement policy studies have proposed the use of multiple cache replacement policies and to alternate between them depending on which policy is best given past memory access behavior. Subramanian et al. presented adaptive caches [17]. Their idea was to use a hardware mechanism that allows the cache to adapt between LRU and Least Frequently Used (LFU) during run-time.

In 2007, Qureshi et al. introduced LRU Insertion Policy (LIP) and Bimodal Insertion Policy (BIP) for L2 cache memories [13]. LIP and BIP are cache memory load strategies designed to operate with an existing LRU or PLRUt mechanism. LIP can reduce Misses per Thousand Instructions (MPKI) by more than 50% compared to LRU. However, in can also increase the MPKI by as much as 90% compared to LRU. The authors in [13] then introduced an adaptive replacement policy to minimize the negative effect of LIP called Dynamic Insertion Policy Set-Dueling (DIP-SD). The DIP-SD policy allows the majority of the cache sets to be adaptive to either LRU or a modified LIP policy called BIP. Their experimental results showed that SD was able to avoid the negative MPKI increases of LIP by dedicating the cache sets to LRU when BIP causes a lot of cache misses.

In the domain of page replacement policy, more sophisticated policies can be implemented as the area and performance constraint of disk storage systems, virtual memory systems, databases etc. are not as demanding as on-chip cache memories. Examples of page replacement policies are LRU-2 [12], 2Q [8], LIRS [7] etc. In [1], the authors stated that each of these algorithms are inferior to the ARC [11] algorithm. ARC was introduced by Megiddo and Modha in 2003. ARC maintains two LRU lists, one to keep track of pages that have been seen only once and the second list keeps track of recent pages that have been seen at least twice. In 2004, Bansal and and Modha introduced CAR (CLOCK with Adaptive Replacement) in [1]. CAR replaces the two LRU lists of ARC with two circular buffers, each utilizing the CLOCK algorithm. In addition, CAR also adds two history lists.

2.1 Our Contribution

The main purpose in the development of the DC replacement policy is to create a cache replacement policy that has lower overhead cost compared to LRU, captures the 'recency' information as per LRU and is scan resistant by reducing cache pollution.

We chose the CLOCK algorithm due to its low overhead cost. The CLOCK algorithm was then modified to be scan resistant. The CLOCK algorithm and the modified CLOCK algorithm are then implemented together using the set dueling methods described in [13] to create an adaptive cache replacement policy, namely Dueling CLOCK. The CLOCK algorithm inherently records the 'recency' information via its reference bit.

In the implementation of DC, the reference bit of the CLOCK algorithm is referred to as the *hitBit*. The 'frequency' feature of memory accesses is also exploited through the use of the *hitBit*. Previously, Robinson et al., in [14], observed that 90% of cache blocks that are loaded into the cache are accessed only once before they are evicted and the remaining 10% of cache blocks that are accessed while they

[1] N refers to cache associativity

```
1: //tagHit - tag comparator result
2: //tagAddr - tag portion of the memory address
3: //cache_Hit - way of the hit cache block when tagHit = TRUE
4: //newData - data fetched from lower level memory
5: //data[k] - cache data-RAM block of way k
6: //tag[k] - cache tag-RAM block of way k
7: //Initialization procedure
8: set replacePtr = 0
9: for i = 0 to cacheAssociativity do
10:     set hitBit[i] = 0;
11: end for
12: //CLOCK replacement algorithm
13: for all cache accesses do
14:     if tagHit == TRUE then
15:         //on cache hit events, where tag[cache_Hit] == tagAddr
16:         set hitBit[cache_Hit] = 1;
17:     else
18:         //on cache miss events
19:         set replacePtr = (replacePtr + 1) mod cacheAssociativity
20:         while hitBit[replacePtr] == 1 do
21:             set hitBit[replacePtr] = 0;
22:             set replacePtr = (replacePtr + 1) mod cacheAssociativity
23:         end while
24:         set data[replacePtr] = newData;
25:         set tag[replacePtr] = tagAddr;
26:     end if
27: end for
```

Figure 2: CLOCK Algorithm

are in the cache have a 99% probability of being accessed again in the future. The *hitBit* allows DC to dynamically identify the cache blocks that have their $hitBit = 0$ as being the 90% of cache blocks that will be accessed only once.

The contributions presented in this paper are as follows,

- We present a scan resistant version of the CLOCK algorithm.
- We present the DC cache replacement policy.
- We provide a hardware implementation of the DC policy for use in on-chip caches.

3. DC Replacement Policy

The DC replacement policy is an adaptive method based on the CLOCK algorithm. To implement the CLOCK algorithm for on-chip cache replacement policy, we created a pointer $replacePtr$ and an array of *hitBit* (reference bits) for each cache block in a cache set.

The CLOCK algorithm uses a circular buffer with the $replacePtr$ pointing to the cache block that is to be replaced when a cache miss occurs. Figure 2 shows the algorithm of the CLOCK algorithm. During a cache hit, the CLOCK algorithm will set the *hitBit* of the accessed cache block to '1' to indicate that the cache block has been hit. When a cache miss occurs, CLOCK will search for a cache block that has its $hitBit = 0$ to store the data currently being fetched from the lower level memory. The CLOCK will first increment the $replacePtr$ then start searching in clockwise direction for the next available cache block that has its $hitBit = 0$. During the search process, the CLOCK algorithm will also reset the *hitBit* of cache blocks that it comes across.

The cache replacement logic for CLOCK policy is shown in Figure 3. Input to the cache replacement logic for CLOCK are the *hitBit* and the $replacePtr$. In Figure 3, the S inputs are the $replacePtr + 1$ (indicating the next cache block in clockwise direction) and the V inputs are the $\overline{hitBit}$. The internal structure of the cache replacement logic for CLOCK is shown in Figure 4, which we call the First One Detector (FOD). The FOD only operates during a cache miss.

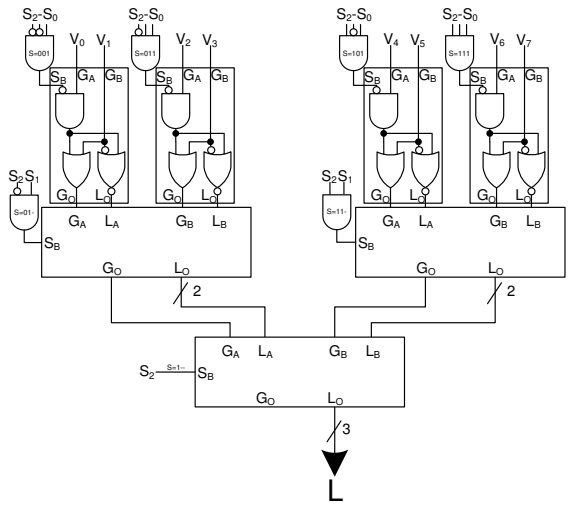

Figure 3: 8-input FOD

In the event of a cache miss, the FOD will determine (according to the algorithm in Figure 2) the address of the data-RAM where the new data should be written. The required functionality is to find the first bit with the logic value '0' in the *hitBit* vector that appears to the right of the bit pointed by the $replacePtr$. To simplify the circuit implementation, we first negate the *hitBit* and use the $\overline{hitBit}$ as input to the FOD. The structure of the FOD is shown in Figure 4.

Figure 3 shows the implementation of the FOD for an 8-bit V. The structure is constructed as a tree of FODUnits (Figure 4). The logic for the L_O output of the top level can be reduced as shown in 3. Each FODUnit covers a range of bits and takes 5 input signals and produces two output signals. The structure takes a divide-and-conquer strategy with pairs of adjacent ranges being merged to calculate the results of the next stage down until the final FODUnit covers the entire range of bits. The outputs operate slightly differently depending on whether S is within the range covered by the unit. If not, then G_O is asserted when at least one '1' is present within the range and L_O provides the relative position of the *first '1'* from the left. If S is within the range covered by the unit, G_O is asserted when there is at least one '1' to the right of position S and in this case, L_O contains the location of the first '1' to the right of position S; otherwise, L_O contains the location of the first '1' from the left of the range covered by the FODUnit. Four of the input signals of each FODUnit come from the outputs of the previous two stages which each operate on half of the bits of the current stage; G_A and L_A being the result of the left half and G_B and L_B being the result of the right half. The final input, S_B, indicates if S is within the range covered by the right half, which is easily calculated via a comparison with the upper bits of S as shown in Figure 4. The result L of the final FODUnit provides the first '1' in V after location S.

To create a scan resistant version of the CLOCK algorithm, we force the $replacePtr$ to not advance to the next cache block whenever the *hitBit* of the cache block pointed by the $replacePtr$ is equal to 'zero' when a cache miss occurs (i.e., we omit line 19 from the CLOCK algorithm shown in Figure 2).

We then create an adaptive cache replacement policy to dynamically choose between the CLOCK algorithm and the scan resistant CLOCK algorithm. The idea of set dueling was first introduced in [13]. The cache sets are divided into three unequal groups. Group 1 is dedicated to a cache replacement policy (the CLOCK algorithm in DC implementation). Group 2 is dedicated to a different cache replacement policy (the scan resistant version of CLOCK algorithm in DC

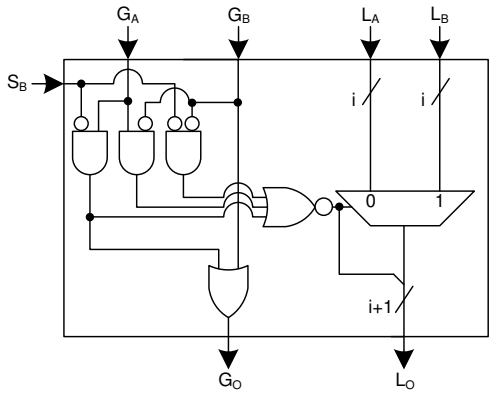

Figure 4: Internals of a FODUnit

replacement policy	access time (ns)		access power (nW)		Area (μm^2)
	hit	miss	hit	miss	
4-way LRU	0.35	0.35	2718	2718	565
4-way PLRUt	0.11	0.17	1263	1878	311
4-way DC	0.11	0.46	1053	2109	889
8-way LRU	0.41	0.41	8617	8617	1791
8-way PLRUt	0.15	0.24	3205	4204	726
8-way DC	0.11	0.57	2106	4073	2451
16-way LRU	0.50	0.50	23590	23590	4484
16-way PLRUt	0.16	0.3	6467	7874	1636
16-way DC	0.11	0.69	4212	5935	4721

Table 1: Cache controller unit access time and access energy cost

implementation). Group 3, which is larger than group 1 and 2, is dynamically interchangeable between the cache replacement policies dedicated to group 1 and 2 based on the value of a 10-bit policy select counter (PSEL). The PSEL counter is decremented each time a cache miss occurs on any of the cache set in group 1 and the PSEL is incremented each time a cache miss occurs on any of the cache set within group 2. Group 3 will then adopt the cache replacement policy of group 1 (CLOCK algorithm) whenever the most significant bit of the PSEL is one, otherwise group 3 will adopt the cache replacement policy of group 2 (scan resistant CLOCK algorithm).

An example of the division of the cache sets into three groups are as follows. For a 16-way 1MB cache with a total of 1024 cache sets. There will be 32 cache sets dedicated to the CLOCK algorithm which will decrement the PSEL each time a cache miss occurs on any of the 32 cache sets utilizing the CLOCK algorithm. The second 32 dedicated cache sets will increment the PSEL each time a cache miss occurs on any of the 32 cache sets utilizing the scan resistant CLOCK algorithm. The rest of the 960 cache sets will use the CLOCK algorithm when the most significant bit (msb) of the 10-bit PSEL is 'one' or use the scan resistant CLOCK algorithm when the msb of the PSEL is 'zero'.

repl. policy	total bits	growth factor
LRU (stack)	$N \times log(N)$	$N \times log(N) + 2N$
LRU (systolic)	$2N \times log(N) + N$	$2N \times log(N) + 5N$
LRU ([10])	$\sum_{i=1}^{log(N)} N - 2^{i-1}$	$N \times log(N) + N$
PLRUt	$N-1$	N
DC	$N + log(N) + 10$	$N+1$

Table 2: Extra RAM required by replacement policies (N = cache associativity)

Application	FastForward (B)	MPKI (LRU)	MPKI (OPT)	% Compulsory Misses
ammp	67.5	2.15	0.23	3.01
applu	86.5	8.70	5.89	48.02
apsi	299.6	4.19	3.35	77.58
art	33.0	12.83	0.31	0.20
bzip2	26.4	1.71	0.58	12.67
crafty	121.9	0.09	0.09	87.12
eon	27.8	0.01	0.01	100.00
equake	81.2	10.91	1.85	7.89
facerec	93.4	3.19	1.00	11.27
fma3d	207.0	3.35	2.58	53.39
galgel	85.8	3.12	0.42	5.83
gap	144.6	5.47	4.26	57.88
gcc	15.3	3.63	0.51	85.79
gzip	36.0	0.08	0.08	99.08
lucas	1.3	3.83	2.92	61.94
mcf	55.3	26.95	2.23	4.89
mesa	73.8	0.29	0.29	77.14
mgrid	99.6	5.02	3.06	35.83
parser	181.2	2.05	1.10	19.66
perlbmk	14.1	0.05	0.05	100.00
sixtrack	253.4	0.18	0.13	15.94
swim	0.6	11.41	7.24	37.44
twolf	110.6	1.51	0.19	2.25
vortex	27.1	0.31	0.28	58.25
vpr	46.8	3.48	0.96	15.32
wupwise	252.1	1.08	0.87	87.76

Table 3: Benchmark Summary (FastForward value is in Billions of instructions)

3.1 Memory Access Cost

Table 1 shows the cache hit access time, cache miss access time and the on-chip area of the combinational logic needed to implement a single-cycle implementation of stack-based LRU [6], PLRUt [6] and the DC policies. Column 1 shows the type of replacement policy and the associativity size, column 2 shows the cache hit access time delay, column 3 shows the cache miss access time delay, column 4 shows the cache hit access dynamic power, column 5 shows the cache miss access dynamic power and column 6 shows the area in μm^2. For comparison purposes, all three circuits were written in VHDL and synthesized with Synopsys Design Compiler [18] using TSMC 90nm technology library [19]. The access power is calculated based on a 50% probability of input activity.

For comparison, a 1MB 8-way cache has an access time of 2.65ns, leakage power per bank of 656mW, access power of 224mW and area cost of $20.2mm^2$ (These numbers were obtained from CACTI 5.3 [20]). To compare the cost of different cache replacement policies, the following assumptions were made:

- the cache memory total access time for a cache hit is the cache access time (as reported in CACTI) plus the cache hit access time of the cache replacement logic.
- the cache memory total access time for a cache miss is the cache access time (as reported in CACTI), plus the lower level memory access time including off-chip memory access time (lower level memory access time is usually many times larger than the on-chip cache access time).
- leakage power is dependent on the total area of the cache memory (total cache area includes the cache SRAM cells area plus the cache replacement logic area).

The total area and access power of the cache replacement logic is negligible in comparison to the area and access power of the SRAM

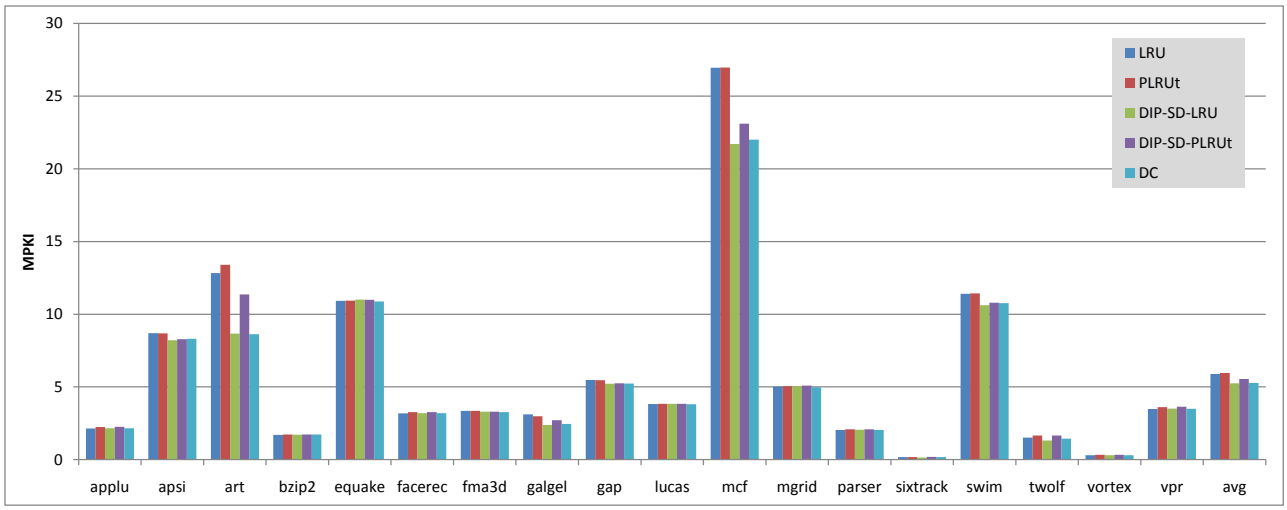

Figure 6: MPKI comparison of a 1MB-16way L2 cache

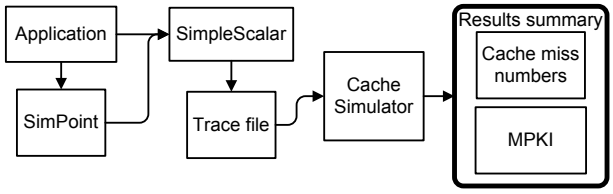

Figure 5: Experimental Setup

cells of cache memory. The cache replacement logic circuits access time, access power and area cost of DC is comparable to the access time, access power and area cost of PLRUt implementation. Thus, we conclude that DC policy implementation is possible for use as on-chip caches as its access time, access power (including its leakage power) and area cost of logic circuits are comparable to the existing PLRUt implementation.

Comparing the access time of the three cache replacement logic circuits for a cache hit, LRU hit access time is approximately three times longer than that of PLRUt. Hence, it is evident why LRU is not used in practice. In comparison, the logic access time of DC is comparable to that of PLRUt.

Table 2 shows the amount of memory bits required for different replacement policies examined in this paper. The amount of memory required by the three implementations of of LRU, PLRUt and DC replacement policies and the growth factor (as the number of associativity doubles) are compared. The numbers shown in Table 2 refer to a single cache set and only counts the amount of memory bits required by the replacement policy. Column 2 of Table 2 shows the amount of memory and column 3 shows the growth factor. DC requires N bits for the $replacePtr$, $log(N)$ bits for the $hitBit$s and a 10bit policy selection counter. As the cache associativity doubles to $2N$, DC requires $N+1$ extra bits. Compared to PLRUt, the replacement policy DC requires an extra $log(N)$ of memory bits per cache set. It should be noted that all three versions of LRU implementation grows at a much faster rate compared to PLRUt or DC, which is another reason why LRU is not implemented for associativities larger than 4.

4. Experimental Setup

Trace-based simulations were performed to evaluate the performance of the DC replacement policy and the results were compared to LRU, PLRUt, DIP-SD-LRU, DIP-SD-PLRUt and DC. Figure 5 displays the experimental methodology. Memory traces were generated using SimPoint [15] and SimpleScalar [3]. Application benchmarks are taken from SPEC2000 [16]. Execution of all twenty six SPEC CPU2000 applications were analyzed using SimPoint in single simulation point mode, with 100 million instructions interval to identify the most relevant simulation points of each application. We then used SimpleScalar to generate trace files containing 300 million instructions (for each application) by fast-forwarding simulation to the point identified by SimPoint.

The L1 cache parameters for instruction and data caches were set to be 2-way associative, 64 Bytes line size and 16KB with LRU replacement policy. The L2 cache is a unified instruction and data cache with associativity ranging from 4-way to 32-way, line size fixed at 64 Bytes, and cache size ranging from 128KB to 4MB.

A cache simulator was built to simulate the replacement policies stated in the first paragraph of this section. The cache simulator was verified against DineroIV [4] and the DIP-SD cache simulator (provided by authors of [13] at http://users.ece.utexas.edu/∼qk/dip/). We compare the MPKI (Misses per thousand instructions) of each application for the different cache replacement policies. Table 3 shows a summary of the individual application within SPEC2000. Column 2 of Table 3 shows the number of billion of instructions fast forwarded to create the memory trace. The value of MPKI for a 1MB 16-way L2 cache is shown for LRU policy and Belady MIN algorithm [2] in column 3 and 4 respectively. Compulsory misses are obtained from simulation using DineroIV [4] for a 16-way 1MB L2 cache with LRU policy. It should be noted that some of the applications show a 100% compulsory miss which should translate to minimal or no difference in the MPKI measurement of different replacement policies. It should be noted that 8 out of the 26 applications have compulsory misses larger than 70% and they have been removed from the results section due to lack of space. These applications are ‘apsi’, ‘crafty’, ‘eon’, ‘gcc’, ‘gzip’, ‘mesa’, ‘perlbmk’ and ‘wupwise’.

5. Results

We compared the Misses per Thousand Instructions (MPKI) of LRU, PLRUt, DIP-SD-LRU [13], DIP-SD-PLRUt [13] and DC. The authors in [13] stated that the concept of set dueling is applicable to both LRU and PLRUt. Hence, we presented results for both DIP-SD-LRU and DIP-SD-PLRUt cache replacement policies. Figure 6

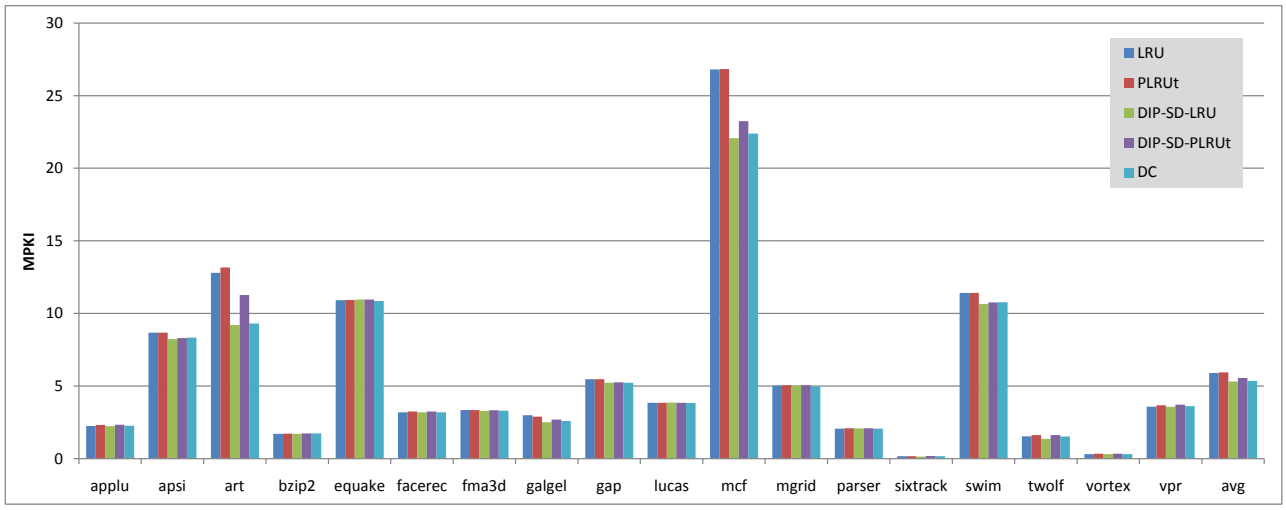

Figure 7: MPKI comparison of a 1MB-8way L2 cache

shows the MPKI comparison for a 16-way 1MB L2 cache of (the bars from left to right) LRU, PLRUt, DIP-SD-LRU, DIP-SD-PLRUt and DC for the 18 benchmarks from SPEC CPU2000. The last sets of bars shows the average MPKI across all 18 applications. The results showed that DC has similar performance when compared to DIP-SD-LRU. However, DIP-SD-LRU has a higher overhead cost compared to the DC policy implementation. Compared to LRU, PLRUt and DIP-SD-PLRUt, DC lowers MPKI by an average of 10.6%, 11.5% and 5.1%, respectively for a 1MB 16-way L2 cache. Figure 7 shows MPKI comparison for a 8-way 1MB L2 cache. The results show DC can reduce MPKI by an average of 9.1%, 9.7% and 3.6% compared to LRU, PLRUt and DIP-SD-PLRUt, respectively. It should be noted that other cache configurations not shown in the paper show similar MPKI reduction results.

6. Conclusions

In this paper, we have presented an algorithm for on-chip L2 cache replacement policy. The contributions of this paper includes a gate level hardware implementation of the CLOCK algorithm for on-chip caches, a scan resistant version of the CLOCK algorithm and the DC cache replacement policy.

Timing and power analysis of the gate level implementation of the CLOCK algorithm have shown that the CLOCK, including the DC policy, is feasible for implementation within an on-chip cache with comparable timing and power cost to PLRUt implementation. The experimental results presented in this paper have shown that for a 16-way 1MB L2 cache, DC has similar MPKI performance compared to DIP-SD-LRU and DC can reduce MPKI by an average of 10.6%, 11.5% and 5.1% when compared to LRU, PLRUt and DIP-SD-PLRUt, respectively.

Acknowledgements

This research was supported in part by the Australian Research Council (ARC) Discovery Project (DP0985168).

7. References

[1] S. Bansal and D. S. Modha, "CAR: Clock with Adaptive Replacement," *USENIX Symposium on File and Storage Technologies,* March, 2004.

[2] L. A. Belady, "A Study of Replacement Agorithms for A Virtual-Storage Computer," *IBM Systems Journal,* pp. 78-101, 1966.

[3] D. C. Burger and T. M. Austin, "The SimpleScalar tool-set, Version 2.0," *Technical Report 1342,* Department of Computer Science, UW, June, 1997.

[4] J. Edler and M. D. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator," *http://www.cs.wisc.edu/~markhill/DineroIV/.*

[5] J. P. Grossman, "A Systolic Array for Implementing LRU Replacement," *MIT-AI Aries Group Technical Memos, (http://www.ai.mit.edu.projects/aries/Documents/Memos/ARIES-18.pdf)* March 2002.

[6] J. Handy, "The Cache Memory Book," *Academic Press,* London, 1998.

[7] S. Jian and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," *ACM SIGMETRICS,* 2002.

[8] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *VLDB,* 1994.

[9] M. Kampe, P. Stenstrom and M. Dubois, "Self-CoFIFOecting LRU Replacement Policies," *Computing Frontiers,* Italy, 2004.

[10] A. J. Loper et al., "Method for Implementing a Four-Way Least Recently Used (LRU) Mechanism in High Performance," *US Patent 5,765,191* 9 June 1998.

[11] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," *2nd USENIX Conference on File and Storage Technologies,* San Francisco, 2003.

[12] E. J. O'Neil, P. E. O'Neil and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *ACM SIGMOD,* pp. 297-306, 1993.

[13] M. K. Qureshi et al., "Adaptive Insertion Policies for High Performance Caching," *ISCA,* San Diego, 2007.

[14] J. T. Robinson and M. V. Devarakonda, "Data Cache Management using Frequency Based Replacement," *SIGMETRICS,* pp.134-142, 1990.

[15] T. Sherwook et al., "Automatically Characterizing Large Scale Program Behavior," *International Conference on Architectural Support for Programming Languages and Operating Systems,* October 2002.

[16] Standard Performance Evaluation Corporation. SPEC2000. *http://www.spec.org.*

[17] R. Subramanian, Y. Smaragdakis and G. H. Log, "Adaptive Caches: Effective Shaping of Cache Behavior to Workloads," *IEEE MICRO 39,* 2006.

[18] Synopsys, Inc., "Design Compiler," *http://www.synopsys.com.*

[19] Taiwan Semiconductor Manufacturing Corporation, "TCBN90G," *TSMC 90nm Core Library Databook,* Oct, 2003.

[20] S. Thoziyoor, N. Muralimanohar, J. H. Ahn and N. P. Jouppi, "CACTI 5.1,"*Technical Report HPL-2008-20,* HP Laboratories, Apr. 2, 2008.