The 3P and 4P cache replacement policies

Pierre Michaud

The 3P and 4P cache replacement policies

Pierre Michaud
INRIA Rennes - Bretagne Atlantique
Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
pierre.michaud@inria.fr

1 Introduction

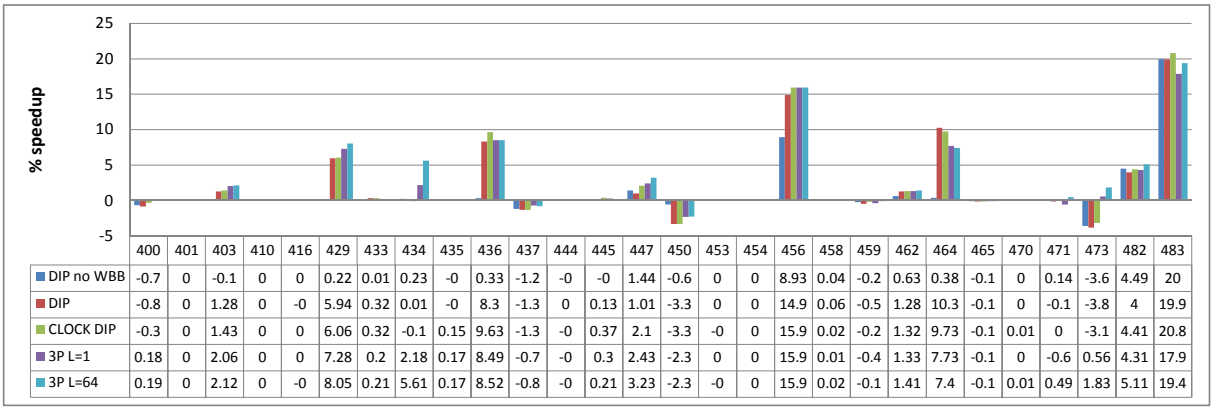
Replacement policies for virtual memories or caches have been studied for almost 5 decades. An optimal *offline* replacement policy was introduced by Belady during the 1960's [2]. It is termed *offline* because it relies on the knowledge of future address references. However, replacement policies that can be implemented in practice are *online* policies that have no knowledge of future references. Actually, there does not exist any optimal online replacement policy, in the sense that it would outperform all the other policies on all address sequences. This can be seen, for instance, by considering the set of all the possible address sequences of a given length, with addresses taken from a finite set. On this set of sequences, *all* the online replacement policies have the same average miss ratio. Nevertheless, not all replacement policies are equivalent in practice. Program-generated address sequences have characteristics that make certain policies better than others in practice. The LRU policy and some approximations of it, like the CLOCK policy, are known to be good replacement policies in many practical situations. On some sequences, LRU is almost as good as Belady's policy. Yet, LRU's success is very dependent on applications characteristics, and there exists some applications for which LRU is far from Belady's minimum. In 1972, Thorington and Irwin proposed a hardware paging mechanism to select dynamically, for each application, the best among a predefined set of policies [7]. This general idea was rediscovered recently for L2 and L3 processor caches [5, 6, 4]. In particular, two key ideas were introduced by Qureshi et al. : *set sampling* [5] and *bimodal insertion* (BIP) [4]. These two ideas are used in the DIP policy [4]. The 3P and 4P replacement policies we propose are based on the CLOCK policy and use set sampling and bimodal insertion. We make four contributions : (1) we show how bimodal insertion can be emulated under the CLOCK policy; (2) we introduce a multi-policy selection mechanism that allows combining more than two different policies; (3) we introduce the 3P policy that uses a bimodal insertion improving memory-level parallelism; (4) we propose the 4P thread-aware policy and a simple method for fairer sharing of a cache by multiple programs.

Our submission to the competition is configured to simulate 3P for the single-core configuration and 4P for the multi-core configuration.

2 The 3P cache replacement policy

2.1 DIP and write-back bypass

The 3P policy is derived incrementally from the DIP policy. DIP uses set sampling and bimodal insertion [4]. Several variations of DIP are possible. The DIP variant that is described in [4] (which we simply call *DIP* in this paper) is based on LRU and assumes that the cache block size is the same for all cache levels. A fraction of the cache sets is dedicated to LRU, an equally sized fraction is dedicated to LRU BIP, and all the other cache sets are *follower* sets using the best policy according to the sign of the PSEL counter which gives the difference between the number of misses in LRU-dedicated sets and in BIP-dedicated sets. Under LRU BIP, a block is promoted to the most-recently-used position upon a hit, or with a probability ε called the *bimodal throttling parameter* [4]. BIP is useful for preventing large data sets from polluting the cache with blocks that will not be reused before eviction. For large associativities, BIP is an effective way to emulate cache bypassing. With DIP, it is important that all cache levels use the same block size. Some recent studies evaluating DIP have assumed L1 blocks smaller than L2 blocks, and this "kills" DIP. This does not mean that the principles behind DIP are incompatible with blocks of different sizes, this means that DIP must be modified and adapted accordingly (some simple solutions are possible, they are out of the scope of this study). Nevertheless, the simulation setup for the competition assumes a unique block size for all cache levels, so we do not have the problem here. An important feature of our DIP implementation is to not update the state of the replacement policy when the L3 cache is accessed by L2 write-backs. We call this feature *write-*



	400	401	403	410	416	429	433	434	435	436	437	444	445	447	450	453	454	456	458	459	462	464	465	470	471	473	482	483
DIP no WBB	-0.7	0	-0.1	0	0	0.22	0.01	0.23	-0	0.33	-1.2	-0	-0	1.44	-0.6	0	0	8.93	0.04	-0.2	0.63	0.38	-0.1	0	0.14	-3.6	4.49	20
DIP	-0.8	0	1.28	0	-0	5.94	0.32	0.01	-0	8.3	-1.3	0	0.13	1.01	-3.3	0	0	14.9	0.06	-0.5	1.28	10.3	-0.1	0	-0.1	-3.8	4	19.9
CLOCK DIP	-0.3	0	1.43	0	0	6.06	0.32	-0.1	0.15	9.63	-1.3	-0	0.37	2.1	-3.3	-0	0	15.9	0.02	-0.2	1.32	9.73	-0.1	0.01	0	-3.1	4.41	20.8
3P L=1	0.18	0	2.06	0	0	7.28	0.2	2.18	0.17	8.49	-0.7	-0	0.3	2.43	-2.3	0	0	15.9	0.01	-0.4	1.33	7.73	-0.1	0	-0.6	0.56	4.31	17.9
3P L=64	0.19	0	2.12	0	-0	8.05	0.21	5.61	0.17	8.52	-0.8	-0	0.21	3.23	-2.3	-0	0	15.9	0.02	-0.1	1.41	7.4	-0.1	0.01	0.49	1.83	5.11	19.4

Figure 1: *Percentage of speedup relative to LRU on a 1MB 16-way L3 cache for DIP without WBB, DIP, CLOCK DIP, 3P with $L = 1$ and 3P with $L = 64$.*

back bypass (**WBB**). WBB was assumed in the original DIP [4], but its importance was not highlighted. The LRU policy giving the baseline performance in this study does not use WBB[1]. Unless specified otherwise, the other policies use WBB. For simulations, we generated traces from the SPEC 2006 benchmarks. Each trace represents 100 millions instructions (the first 40 billions instructions in each benchmark were skipped). Our simulation results were obtained with the simulation infrastructure provided for the competition. Figure 1 shows, for each benchmark[2], the speedup relative to LRU on a 16-way 1MB L3 cache. DIP uses a 12-bit PSEL counter, 2×32 dedicated sets, and a 6-bit BIPCTR counter to implement $\varepsilon = 1/64$, as described in [4] (BIPCTR is incremented on each miss and we insert the missing block in the MRU position when BIPCTR is null). First, we observe that DIP without WBB provides significant speedup on a few benchmarks (*456.hmmer*,*482.sphinx3*,*483.xalancbmk*), as expected. Second, we observe that WBB provides yet more speedup, especially on *429.mcf*, *436.cactusADM*, *456.hmmer* and *464.h264ref*.

2.2 The CLOCK DIP policy

The 3P policy is based on a variant of CLOCK. In practice, CLOCK needs less storage than LRU and it consumes less energy on cache hits. Variants of the CLOCK paging policy have been known for a long time [3, 1], and its use as an L2 and L3 cache replacement policy was revisited recently [8]. The CLOCK variant that we implemented works as follows. Each block in the 16-way set-associative L3 cache is associated with a *use* bit, and there is one 4-bit clock *hand* stored along with each cache set. The storage overhead of CLOCK is $4 + 16 \times 1 = 20$ bits per cache set, which is less than what practical LRU implementations require. Upon a cache hit, the *use* bit associated with the hitting block is set. Upon a cache miss, and for choosing a victim, we inspect the use bit of the block stored in the way pointed to by the hand. If the use bit is null, the block is victimized. Otherwise, the use bit is reset, the hand is incremented modulo 16, and the same task is repeated as many times as necessary until we find a victim (in the worst case, 16 iterations are necessary[3]). When the victim way is found, we insert the newly allocated block *B* there. There are two possible options : either we leave the use bit for that way reset, or we may set it. If we set the use bit for *B*, the hand will be incremented on the next miss, and block *B* will likely be protected against eviction for several misses, in a LRU-like fashion. If we choose not to set the use bit for *B*, then the hand will be incremented on the next miss only if block *B* is re-referenced in the meantime. If not re-referenced, block *B* will be the next victim, in a BIP-like fashion. In the **CLOCK DIP** policy, some sets are dedicated to the normal CLOCK policy that sets the use bit on a miss, some sets are dedicated to the **CLOCK BIP** policy that sets the use bit with probability $\varepsilon = 1/64$ on a miss, and the other sets use the best among CLOCK and CLOCK BIP. Figure 1 shows speedups for CLOCK DIP. The parameters (PSEL width, dedicated sets, BIPCTR) are the same as for DIP. In practice, CLOCK DIP is very close to DIP.

2.3 The 3P policy

On benchmark *473.astar*, we observed some address patterns where the accesses are approximately cyclic, but the active working set drifts progressively, some addresses

[1] In our simulations, WBB is almost effectless with LRU.

[2] We do not have results for *481.wrf* because we could not compile it.

[3] The L3 miss latency is high, there should be enough time to perform this task sequentially. Pure combinational logic may also be used.

being no longer accessed after some time and new addresses entering the active working set. On this kind of pattern, BIP with $\varepsilon = 1/64$ underperforms because of the working-set drifting, and LRU underperforms because of the (approximately) cyclic patterns. In such situation, we found that BIP with $\varepsilon = 1/2$ is a slightly better policy. This led us to the 3P policy, where some sets are dedicated to CLOCK, some to CLOCK BIP with $\varepsilon = 1/64$, and some to CLOCK BIP with $\varepsilon = 1/2$. However, a single PSEL counter is not sufficient to choose the best out of 3 different policies, so we had to use a different mechanism, inspired from the PSEL counter. To choose between N policies, we use N counters $P_1, \cdots, P_N$. When a miss occurs in a set dedicated to policy j, we add $N-1$ to P_j and we subtract 1 to each of the other P_i, $i \neq j$. By doing this, the sum of all P_i's is always zero, and $P_j - P_k$ equals N times the difference between the number of misses in sets dedicated to policies j and k. The best policy j is the one whose P_j has the smallest value. Each counter P_i has a finite width, and we must deal with saturation carefully : either we update all the counters simultaneously (no saturation), or we update no counter at all.

2.4 Improving memory-level parallelism

The way misses are distributed in time may have an impact on performance. For the same total number of misses, and if there is enough memory bandwidth, it is generally better to have misses clustered than to have misses uniformly distributed in time. Clustering misses on an out-of-order core increases the chance that two misses overlap, so that the second miss latency can be hidden. In CLOCK DIP, we use Qureshi et al.'s method : a 6-bit BIPCTR counter is incremented on each miss and the use bit is set for the missing block when BIPCTR is null. For 3P, we experimented with a new implementation for the bimodal throttling parameter. We introduce a parameter L. In a cache set under CLOCK BIP with bimodal throttling ε, the BIPCTR counter is $\log_2(L/\varepsilon)$ bit wide, and we set the use bit on a miss when the BIPCTR value is less than L. Note that the case $L = 1$ is identical to Qureshi et al.'s method [4]. Figure 1 shows the speedup for 3P with $L = 1$ and $L = 64$. For 3P, we have 3×16 dedicated sets, and counters P_1, P_2 and P_3 are 11 bits each. We observe that $L = 64$ yields a higher speedup than $L = 1$ on a few benchmarks, especially *434.zeusmp* (+3.4%), *473.astar* (+1.3%) and *483.xalancbmk* (+1.3%). Yet, on these benchmarks, the total number of misses is almost unchanged between $L = 1$ and $L = 64$ (for 473 and 483, the number of misses is even slightly increased with $L = 64$). Miss clustering seems to be more effective for $\varepsilon = 1/2$ than for $\varepsilon = 1/64$.

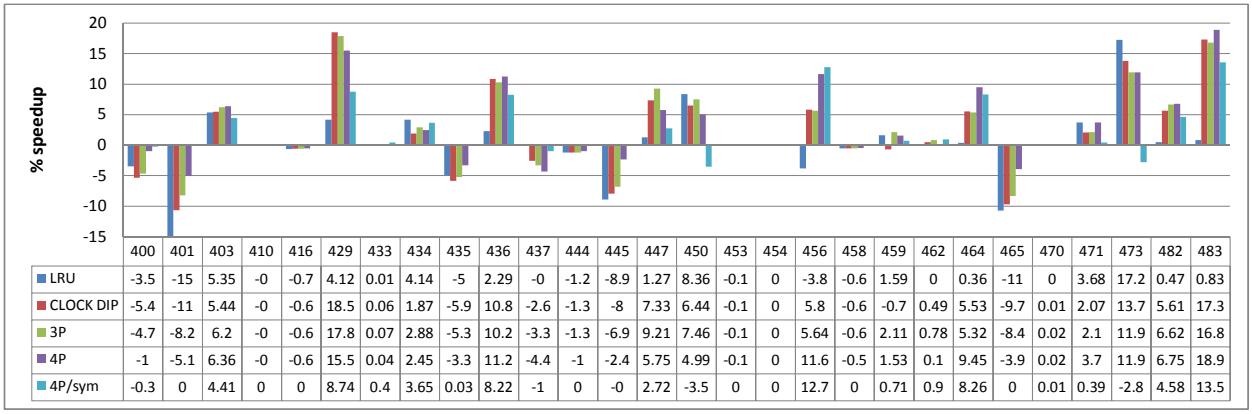
3 The 4P cache replacement policy

DIP is effective not only for a private L2/L3 cache but also for a cache shared by multiple programs/threads. In a shared cache, DIP can decrease the number of misses for BIP-friendly threads that have a large working set accessed in a cyclic way. But DIP may also decrease the number of misses for a LRU-friendly thread that would not benefit from DIP in a private cache, by decreasing the rate at which blocks belonging to that thread are evicted by other threads. Hence a thread-unaware policy like DIP or 3P is on average better than LRU not only in a private L3 cache but also in a shared L3 cache. For evaluating the shared cache, we used the simulation infrastructure provided for the competition and we used a *random* workload consisting of 100 4-thread combinations, where each combination consists of 4 benchmarks taken randomly in our list of 28 SPEC 2006 benchmarks (the same benchmark may appear several times in the same 4-thread combination). We have a total of $100 \times 4 = 400$ CPI values. We define the **average CPI** for a benchmark as the arithmetic mean of all the CPI values obtained for that benchmark in the list of 400 values[4]. We define the **symmetric CPI** of a single-thread benchmark as the arithmetic mean of the 4 CPI values obtained when running 4 instances of that benchmark simultaneously, what we call a *symmetric run*. For each of the 28 SPEC 2006 benchmarks, we define the **baseline CPI** as the symmetric CPI under LRU. The symmetric performance is an interesting metric because a symmetric run is not an unlikely situation. Moreover, the symmetric performance may be viewed as a *fair* performance value, i.e., the minimum performance to expect for a sequential application. For a given policy, we define the speedup of a single-thread benchmark as the baseline CPI for that benchmark divided by its average CPI on the random workload. Figure 2 shows the speedups for LRU, CLOCK DIP and 3P on a 4MB 16-way shared L3 cache. For CLOCK DIP we use 2×64 dedicated sets and $\varepsilon = 1/32$. For 3P, we use 3×64 dedicated sets, $\varepsilon = 1/32$, $L = 64$, and 4 BIPCTR counters (one per thread). It can be observed that LRU is not a fair policy, as the speedups for some benchmarks (e.g., *401.bzip2* and *465.tonto*) are markedly negative. CLOCK DIP and 3P increase the performance of a majority of benchmarks. Note that *401.bzip2* benefits from bimodal insertion even though it is not itself BIP-friendly.

3.1 The 4P policy

Even with 3P, some benchmarks still do not obtain a fair share of the cache capacity. These are typically "fragile" threads that have a low miss rate, i.e., a small number of

[4] We checked that each benchmark has at least 8 CPI values.



	400	401	403	410	416	429	433	434	435	436	437	444	445	447	450	453	454	456	458	459	462	464	465	470	471	473	482	483
LRU	-3.5	-15	5.35	-0	-0.7	4.12	0.01	4.14	-5	2.29	-0	-1.2	-8.9	1.27	8.36	-0.1	0	-3.8	-0.6	1.59	0	0.36	-11	0	3.68	17.2	0.47	0.83
CLOCK DIP	-5.4	-11	5.44	-0	-0.6	18.5	0.06	1.87	-5.9	10.8	-2.6	-1.3	-8	7.33	6.44	-0.1	0	5.8	-0.6	-0.7	0.49	5.53	-9.7	0.01	2.07	13.7	5.61	17.3
3P	-4.7	-8.2	6.2	-0	-0.6	17.8	0.07	2.88	-5.3	10.2	-3.3	-1.3	-6.9	9.21	7.46	-0.1	0	5.64	-0.6	2.11	0.78	5.32	-8.4	0.02	2.1	11.9	6.62	16.8
4P	-1	-5.1	6.36	-0	-0.6	15.5	0.04	2.45	-3.3	11.2	-4.4	-1	-2.4	5.75	4.99	-0.1	0	11.6	-0.5	1.53	0.1	9.45	-3.9	0.02	3.7	11.9	6.75	18.9
4P/sym	-0.3	0	4.41	0	0	8.74	0.4	3.65	0.03	8.22	-1	0	-0	2.72	-3.5	0	0	12.7	0	0.71	0.9	8.26	0	0.01	0.39	-2.8	4.58	13.5

Figure 2: *Percentage of speedup on a 4MB 16-way L3 cache shared by 4 threads. The baseline performance for each benchmark is the symmetric performance under LRU.*

misses per clock cycle. When a fragile thread shares the cache with some "aggressive" threads having a high miss rate, the miss rate of the fragile thread is relatively high to compensate the rate at which its blocks are evicted by the aggressive threads. A possible way to improve the situation is to use bimodal insertion for aggressive threads and normal insertion for fragile threads. We define a thread-aware bimodal insertion policy called **CLOCK TUBIP** : for non-fragile threads, the use bit is set on a miss with probability $\varepsilon = 1/32$; for fragile threads, the use bit is always set on a miss. The 4P policy has 4 kinds of dedicated sets, for CLOCK, CLOCK BIP with $\varepsilon = 1/32$, CLOCK BIP with $\varepsilon = 1/2$ and CLOCK TUBIP. For selecting the best policy, we use 4 counters P_i. To identify fragile threads, we use the following heuristic. We have 4 TMISS counters, one for each thread. The TMISS counters are updated in a way very similar to the counters P_i : when a miss occurs for thread j in a set dedicated to CLOCK TUBIP, we add 3 to the TMISS counter of thread j and we subtract 1 to the TMISS counters of all the other threads[5]. We define a fragile thread as a thread whose TMISS counter value is negative. Results for 4P are shown in Figure 2 (each counter P_i is 11 bits, each TMISS counter is 14 bits, we have 4×64 dedicated sets, and $L = 64$). We give the speedups for 4P on random workloads and on symmetric workloads. Compared to 3P, 4P decreases the performance of aggressive threads like 429, 447 and 450. On the other hand, it increases the performance of fragile threads like 400, 401, 445, 456 and 465. 4P tends to increase the performance of threads that are below the symmetric performance. In that sense, 4P is fairer than 3P.

[5] In a real implementation, the update of TMISS counters should be a function of the actual number of threads running simultaneously. Here we assume 4 running threads.

4 Storage cost

Policies 3P and 4P require 20 bits per cache set in a 16-way cache (one 4-bit hand per set, 1 use bit per block). The 3P version submitted to the competition uses 11-bit P_i counters and one 12-bit BIPCTR counter ($\varepsilon = 1/64$, $L = 64$). The total storage for a 1 MB cache is $1024 \times 20 + 3 \times 11 + 12 = 20525$ bits. The 4P version submitted to the competition uses four 11-bit P_i counters, four 11-bit BIPCTR counters ($\varepsilon = 1/32$, $L = 64$), and four 14-bit TMISS counters. The total storage for a 4MB cache is $4096 \times 20 + 4 \times 11 + 4 \times 11 + 4 \times 14 = 82064$ bits.

References

[1] Y. Bard. Characterization of program paging in a time-sharing environment. *IBM Journal of Res. & Dev.*, 17(5):387–393, 1973.

[2] L.A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(5):78–101, 1966.

[3] F.J. Corbató. A paging experiment with the MULTICS system. In *In Honor of P. M. Morse*, pages 217–228. MIT Press, 1969.

[4] M. Qureshi, A. Jaleel, Y.N. Patt, S.C. Steely Jr, and J. Emer. Adaptive insertion policies for high performance caching. In *Proc. of the 34th Int. Symp. on Computer Architecture*, 2007.

[5] M.K. Qureshi, D.N. Lynch, O. Mutlu, and Y.N. Patt. A case for MLP-aware cache replacement. In *Proc. of the 33rd Int. Symp. on Computer Architecture*, 2006.

[6] R. Subramanian, Y. Smaragdakis, and G.H. Loh. Adaptive caches : effective shaping of cache behavior to workloads. In *Proc. of the 39th Int. Symp. on Microarchitecture*, 2006.

[7] J.M. Thorington and J.D. Irwin. An adaptive replacement algorithm for paged-memory computer systems. *IEEE Trans. on Computers*, C-21(10):1053–1061, October 1972.

[8] J. Zebchuk, S. Makineni, and D. Newell. Re-examining cache replacement policies. In *Proc. of the IEEE Int. Conf. on Computer Design*, 2008.