# Optimal Cache Replacement Policy for Matrix Multiplication

Nenad Anchev, Marjan Gusev, Sasko Ristov, and Blagoj Atanasovski

Ss. Cyril and Methodious University, Faculty of Computer Science and Engineering,
Rugjer Boshkovikj 16, 1000 Skoipje, Macedonia
nenad_ancev@hotmail.com, marjan.gushev@finki.ukim.mk,
sashko.ristov@finki.ukim.mk, blagoj.atanasovski@gmail.com

**Abstract.** Matrix multiplication is compute intensive, memory demand and cache intensive algorithm. It performs $O(N^3)$ operations, demands storing $O(N^2)$ elements and accesses $O(N)$ times each element, where $N$ is the matrix size. Implementation of cache intensive algorithms can achieve speedups due to cache memory behavior if the algorithms frequently reuse the data. A block replacement of already stored elements is initiated when the requirements exceed the limitations of cache size. Cache misses are produced when data of replaced block is to be used again. Several cache replace policies are proposed to speedup different program executions.
In this paper we analyze and compare two most implemented cache replacement policies First-In-First-Out (FIFO) and Least-Recently-Used (LRU). The results of the experiments show the optimal solutions for sequential and parallel dense matrix multiplication algorithm. As the number of operations does not depend on cache replacement policy, we define and determine the average memory cycles per instruction that the algorithm performs, since it mostly affects the performance.

**Keywords:** FIFO, HPC, LRU, Performance, Speedup

## 1 Introduction

CPU runs a particular program by accessing data from the memory, executing basic operations addition or multiplication and storing the results in the memory. The main bottleneck in the process is the data access in memory which is approximately up to 1000 times slower than floating point operation execution [7]. Introducing memory hierarchy based on caches in CPU speeds up the execution of programs that reuse the same data, i.e. cache intensive algorithms. This paper focuses on dense matrix multiplication algorithm.

Most modern multiprocessors use three layer $n$-way associative cache memory to speedup main memory access. The cache size grows but the access time and miss penalty rise going from the lowest L1 to L3 cache. The effect of exploiting last level shared cache affinity is considerable, due its sharing among multiple threads and high reloading cost [14]. Intel introduces Intel Smart Cache into their newest CPUs to improve their performance [8].

However, cache memory speeds up the execution only when matrix data fits in the cache. When the problem size exceeds a particular cache size then the cache misses start generating and the performance decreases. Two drawbacks appear in this case [5]. If a cache block is replaced after a particular matrix element is accessed then the next access to this element will generate a cache miss. The second drawback refers to a situation when there is an access to another matrix element from the same cache block (line) but in meantime the block was replaced. Inefficient usage of cache is possible if matrix elements map onto a small group of same cache sets and initiate a significant number of cache misses due to cache associativity [17].

Cache replacing policy also impacts the algorithm performance, i.e. which cache line will be replaced from some cache set to place the requested data from some of next level caches or main memory into the cache that generated miss. Three basic cache replacement policies are suggested: Random, Least-Recently-Used (LRU) and First-In-First-Out (FIFO) [7]. Several improvements are proposed for LRU. LRU Insertion Policy (LIP) places the incoming line in the LRU position instead of the MRU [15]. The authors in [2] propose even better replacement policy, i.e. a Score-Based Memory Cache Replacement Policy. Adaptive Subset Based Replacement Policy for High Performance Caching is proposed in [6], i.e. to divide one cache set into multiple subsets and victims should be always taken from one active subset when cache miss occurs. Map-based adaptive insertion policy estimates the data reuse possibility on the basis of data reuse history [9]. The authors in [11] propose Dueling CLOCK cache replacement policy that has low overhead, captures recency information in memory accesses and exploits the frequency pattern of memory accesses compared to LRU. A new replacement algorithm PBR_L1 is proposed for merge sort which is better than FIFO and LRU [3]. The authors in [12] propose LRU-PEA replacement policy that enables more intelligent replacement decisions due to the fact that some types of data are less commonly accessed depending on which bank they reside in. The authors in [10] propose cache replacement using re-reference interval prediction to outperform LRU in many real world game, server, and multimedia applications. However, improving replacing policies requires either additional hardware or modification of existing. PAC-PLRU replacing policy utilizes the prediction results generated by the existing stride prefetcher and prevents these predicted cache blocks from being replaced in the near future [18].

In this paper we focus on two most common cache replacement policies, LRU and FIFO and their impact on dense matrix multiplication performance. A tool for automatic computation of relative competitive ratios for a large class of replacement policies, including LRU, FIFO, and PLRU can be found in [16]. LRU has a gap of 50% optimal replacement policies [1]. We realize series of experiments for sequential and parallel execution of dense matrix multiplication on different hardware infrastructure with LRU and FIFO cache replacement policies. The rest of the paper is organized as follows. Section 2 describes the dense matrix multiplication algorithm and its parallel implementation and Section 3 the hardware infrastructure and runtime environment for the experiments. In Sec-

tion 4 we present the results of the experiments and analyze which replacement policy is better for both for sequential and parallel dense matrix multiplication algorithm. Section 5 is devoted to conclusion and future work.

## 2 The Algorithm

We choose squared matrices with dimension $N$. For all $i, j = 0, 1, \ldots, N-1$ the result product matrix $C_{N \cdot N} = [c_{ij}]$ is defined in (1) by multiplying the multiplier matrix $A_{N \cdot N} = [a_{ij}]$ and the multiplicand matrix $B_{N \cdot N} = [b_{ij}]$. More details about algorithm complexity are given in [4]. To exploit maximum performance for parallel execution on $P$ processing elements we use dynamic schedule directive of OpenMP with $chunk = 1$ [13].

$$c_{ij} = \sum_{k=0}^{N-1} a_{ik} \cdot b_{kj} \tag{1}$$

### 2.1 Algorithm Definitions and Analysis

In this section we analyze the algorithm execution. For better presentation and analysis we use CPU clock cycles instead of execution time. Relation (2) derives the total execution clock cycles ($TC$) as a sum of clock cycles needed for operation execution ($CC$) and clock cycles needed for accessing the matrix elements ($MC$) [7].

$$TC = CC + MC \tag{2}$$

$CC$ does not depend neither of CPU architecture nor cache size, associativity and replacement policy, but directly depends of matrix size $N$. CPU executes $N^3$ sums and $N^3$ multiplications or total $2 \cdot N^3$ floating points operations. $MC$ is more interesting for analysis. It depends on matrix size $N$, but also on cache size, associativity and replacement policy.

More important parameters for analysis are the average values of $TC$, $MC$ and $CC$ defined in the next three definitions.

**Definition 1 (Average Total Cycles Per Instruction)** *$CPI_T(N)$ for particular matrix size $N$ is defined as a ratio of total number of clock cycles and total number of instructions given in (3).*

$$CPI_T(N) = \frac{TC}{2 \cdot N^3} \tag{3}$$

**Definition 2 (Average Memory Cycles Per Instruction)** *$CPI_M(N)$ for particular matrix size $N$ is defined as a ratio of total number of memory cycles and total number of instructions given in (4).*

$$CPI_M(N) = \frac{MC}{2 \cdot N^3} \tag{4}$$

**Definition 3 (Average Calculation Cycles Per Instruction)** $CPI_C(N)$ *for particular matrix size* $N$ *is defined as a ratio of total number of calculation cycles* $CC$ *and total number of instructions given in (5).*

$$CPI_C(N) = \frac{CC}{2 \cdot N^3} \tag{5}$$

We measure speed, $TC$, $CC$, $MC$ for each matrix size, number of cores in defined testing environments. We calculate $CPI_T(N)$, $CPI_M(N)$ and $CPI_C(N)$ and analyze the distribution of $CPI_M(N)$ in $CPI_T(N)$. All the experiments are realized both for sequential and parallel execution.

### 2.2 Measurement Methodology

This section describes how we measure $TC$, $CC$, $MC$ to calculate $CPI_T(N)$, $CPI_M(N)$ and $CPI_C(N)$. We measure total execution time $TT$ for each experiment with algorithm described in (1) and then calculate $TC$ as defined in [7] and calculate $CPI_T(N)$ using (3).

To measure $MC$ we developed another algorithm defined in (6). This algorithm performs the same floating point operations on constant operands and writes the results in matrix $C$ elements. The difference is that it does not read from memory or some cache the elements of matrices $A$ and $B$.

$$c_{ij} = \sum_{k=0}^{N-1} a \cdot b \tag{6}$$

Executing this algorithm we measure its execution time $CT$ for each experiment and then calculate the difference from $TC$ and $CT$. Then we calculate $MC$ as defined in [7] using CPU speed for particular processor and calculate $CPI_M(N)$ using (4).

$CC$ and $CPI_C(N)$ are calculated as defined in (7) and (5).

$$CC = TC - MC \tag{7}$$

## 3 The Testing Environment

Two servers with different CPUs with different cache replacement policies are used: FIFO and LRU. Both servers are installed with Linux Ubuntu 10.10. C++ with OpenMP support is used for parallel execution.

FIFO testing hardware infrastructure consists of one Intel(R) Xeon(R) CPU X5680 @ 3.33GHz and 24GB RAM. It has 6 cores, each with 32 KB 8-way set associative L1 and 256 KB 8-way set associative L2 cache. All 6 cores share 12 MB 16-way set associative L3 cache. Each experiment is executed using different matrix size $N$ for different number of cores from 1 to 6. Tests are performed by unit incremental steps for matrix size and number of cores.

LRU testing hardware infrastructure consists of one CPU Quad-Core AMD Phenom(tm) 9550. It has 4 cores, each with 64 KB 2-way set associative L1 and 512 KB 16-way set associative L2 cache. All 4 cores share 2 MB 32-way set associative L3 cache. Each experiment is executed using different matrix size $N$ on different number of cores from 1 to 4. Tests are performed by unit incremental steps for matrix size and number of cores.

## 4 Results of the Experiments

This section presents and compares the results of the experiments on two CPUs with different replacement policies.

### 4.1 Results for CPU with FIFO Cache Replacement Policy

Figure 1 depicts the results of measured speed. $SpeedT(i)$ denotes the speed in gigaFLOPS for algorithm execution on $i$ cores where $i = 1, 2, ..., 6$.

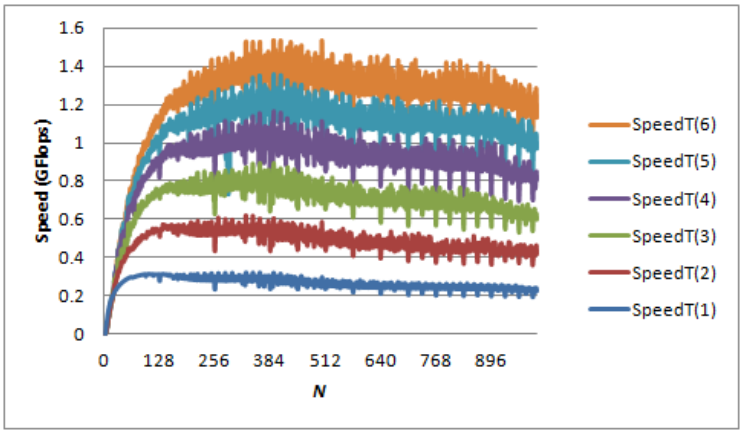


**Fig. 1.** Speed for execution on FIFO CPU.

$CPI_T(N)$ presents another perspective of the experiment. Figure 2 depicts the results for algorithm execution on $1, 2, ..., 6$ cores for each matrix size $128 < N < 1000$. We can conclude that executing the dense matrix multiplication algorithm on more cores needs more average cycles per core for each matrix size $N$. Also, the speed decreases by increasing the matrix size $N$.

The next experiment analyzes the decomposition of the average total cycles per instruction on average calculation cycles per instructions and average memory cycles per instruction. Figure 3 depicts the decomposition of $CPI_T(N)$ on $CPI_M(N)$ and $CPI_C(N)$ for sequential execution. The left graph depicts the

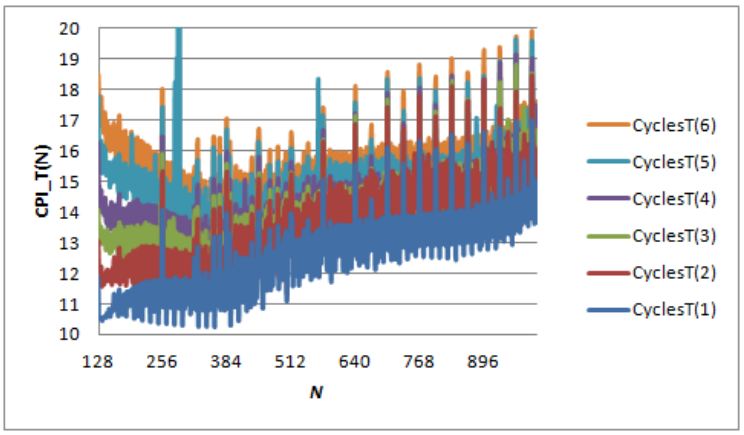


**Fig. 2.** $CPI_T(N)$ for execution on FIFO CPU.

absolute decomposition of $CPI_T(N)$. The conclusion is that $CPI_C(N)$ is almost constant with average value of 4.93 cycles per instruction. More important is that $CPI_M(N)$ follows $CPI_T(N)$, i.e. $CPI_T(N)$ depends directly of average memory cycles per instruction. The right graph depicts the relative value of $CPI_M(N)$ to $CPI_T(N)$. $CPI_M(N)$ has a trend to equalize with $CPI_T(N)$ as $N$ grows.

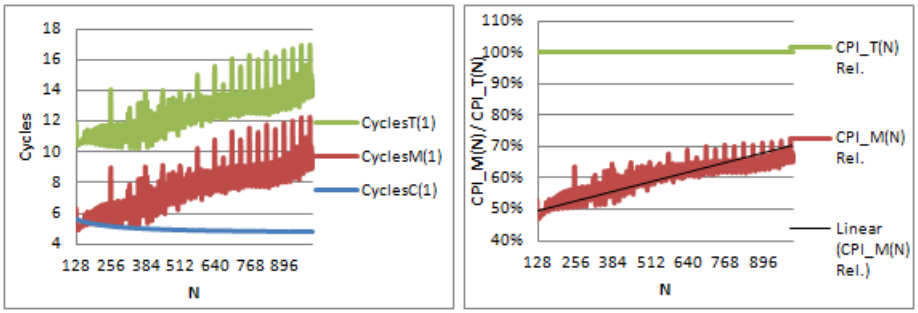


**Fig. 3.** Decomposed $CPI_T(N)$ for sequential execution on FIFO CPU, absolute (left) and relative (right)

### 4.2 Results for CPU with LRU Cache Replacement Policy

Figure 4 depicts the results of measured speed. $SpeedT(i)$ denotes the speed in gigaFLOPS for algorithm execution on $i$ cores where $i = 1, 2, ..., 6$. We can conclude that there is a huge performance drawback after $N > 362$ which is

entrance in the L4 region, i.e. the region where elements of matrices $A$ and $B$ cannot be placed in L3 cache and thus producing L3 cache miss.

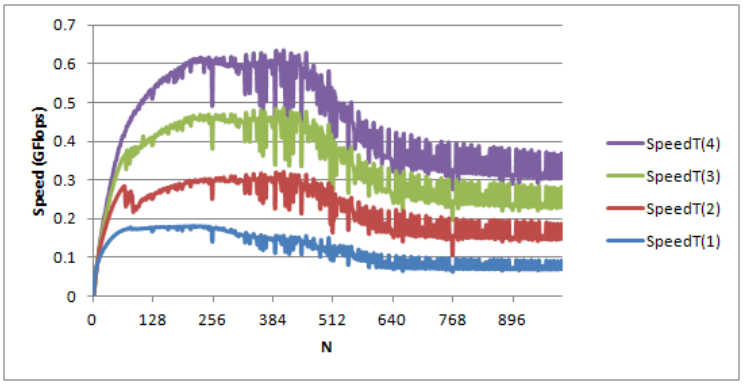


**Fig. 4.** Speed for execution on LRU CPU.

$CPI_T(N)$ presents better the information. Figure 5 depicts results for executions on $1, 2, 3$ and $4$ cores for each matrix size $128 < N < 1000$. We can see 2 regions, Region 1 for $N < 362$ and Region 2 for $N > 362$. The former presents the L1 and L2 cache regions, i.e. dedicated per core regions where matrices can be stored completely in L1 and L2 caches correspondingly. In this region sequential execution provides the worst $CPI_T(N)$ compared to parallel execution. The latter presents L3 and L4 regions, i.e. shared memory regions where matrices can and cannot be stored completely in L3 cache correspondingly. In this region sequential execution provides the best $CPI_T(N)$ compared to parallel execution.

Figure 6 depicts the decomposition of $CPI_T(N)$ on $CPI_M(N)$ and $CPI_C(N)$ for sequential execution. The left graph depicts the absolute decomposition of $CPI_T(N)$. $CPI_C(N)$ is almost constant to the average value of 7.17 cycles per instruction. More important is that $CPI_M(N)$ follows $CPI_T(N)$, i.e. $CPI_T(N)$ depends directly of average memory cycles per instruction. The right graph depicts the relative value of $CPI_M(N)$ to $CPI_T(N)$. As depicted, $CPI_M(N)$ has a trend to equalize with $CPI_T(N)$ as $N$ grows for $N \cdot (N + 1) < 2MB$. This is the case when matrix $B_{N.N}$ and one row od matrix $A_{1.N}$ can be placed in the L3 cache. $CPI_M(N)$ relative remains constant for greater $N$.

### 4.3 LRU and FIFO Cache Replacement Policy Comparison

In this section we compare the results between performance of FIFO and LRU cache replacement policies.

**Speed Comparison** Comapring figures 1 and 4 we can conclude that both infrastructures have a region around entrance to L3 region when the speed begins

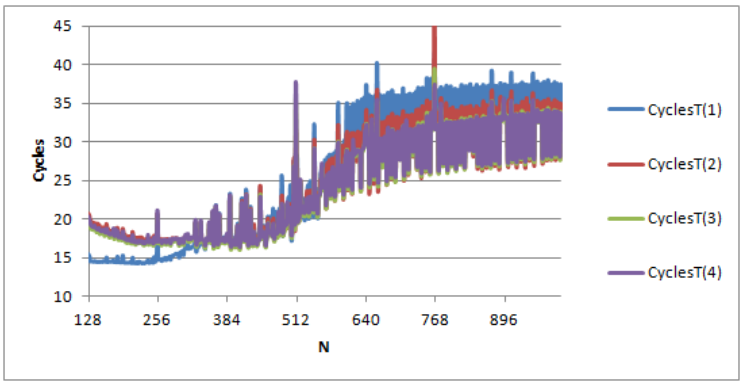


**Fig. 5.** $CPI_T(N)$ for execution on LRU CPU.

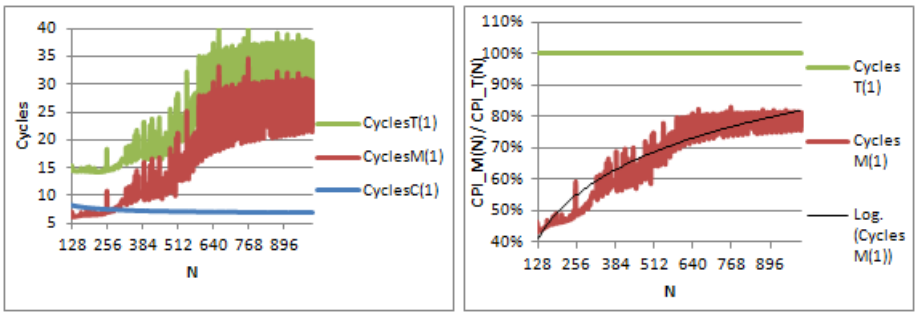


**Fig. 6.** Decomposed $CPI_T(N)$ for sequential execution on LRU CPU, absolute (left) and relative (right)

to fall down to a local maximum. The graphs show that the speed decrease is more emphasized in LRU rather than FIFO. However, it is because L3 region in LRU begins for $N > 362$ and for FIFO CPU for $N > 886$. Therefore the real comparison should be the regions $N > 362$ on LRU CPU with $N > 886$ on FIFO CPU, which are the beginning of L4 region.

***CPI*$_T(N)$ Comparison** Comparing figures 2 and 5 we can conclude that both infrastructures have similar curves for $CPI_T(N)$ for particular region. The important conclusion is that FIFO CPU needs more cycles per core for each matrix size N regardless of cache region (dedicated or shared). However, the LRU CPU has different features. Sequential execution has the best $CPI_T(N)$ in dedicated per core L1 and L2 regions and parallel execution on greater number of cores in shared L3 and L4 regions.

***$CPI_T(N)$* Decomposition Comparison** Comparing figures 3 and 6 (left) we can conclude that both infrastructures have similar curves for $CPI_T(N)$. The graphs show that $CPI_T(N)$ is greater in LRU than FIFO. However, the real comparison should be the regions $N > 362$ on LRU CPU with $N > 886$ on FIFO CPU as explained in the previous subsection. $CPI_M(N)$ is almost parallel compared to $CPI_T(N)$ for all matrix size $N$ in both infrastructures. Also, the similar result is the fact that $CPI_C(N)$ is almost constant for each matrix size $N$ for both CPUs.

***$CPI_M(N)$* Comparison** Comparing figures 3 and 6 (right) we can conclude that $CPI_M(N)$ is relative more closer to $CPI_T(N)$ in LRU than FIFO. However, it is because L3 region in LRU begins for $N > 362$ and for FIFO CPU for $N > 886$. Therefore the real comparison should be the regions $N > 362$ on LRU CPU with $N > 886$ on FIFO CPU, which are the beginning of L4 region and the relative values in LRU CPU are better than FIFO CPU. LRU CPU has average of 59.77% in the region of $N = 362$ and FIFO CPU has average 65.84% in the region of $N = 886$.

## 5 Conclusion and Future Work

In this paper we determine that both cache replacement policies provide similar speed and average cycles per instruction $CPI_T(N)$ for sequential and parallel execution. However, the results show that LRU replacement policy provides best $CPI_T(N)$ for sequential execution in dedicated per core cache memory. Parallel execution provides the best $CPI_T(N)$ in shared memory LRU CPU, i.e. LRU produces greater speedup than FIFO and is more appropriate rather than FIFO cache replacement policy for dense matrix multiplication algorithm.

Our plan for future work is to analyze the performance of other cache replacement policies for sequential and parallel execution, as well as other compute intensive, memory demanding and cache intensive algorithms. With appropriate simulator we can compare different replacement policies with the same cache size, associativity and cache levels.

## References


1. Al-Zoubi, H., Milenkovic, A., Milenkovic, M.: Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In: Proceedings of the 42nd annual Southeast regional conference. pp. 267–272. ACM-SE 42, ACM, New York, NY, USA (2004)
2. Duong, N., Cammarota, R., Zhao, D., Kim, T., Veidenbaum, A.: SCORE: A Score-Based Memory Cache Replacement Policy. In: Emer, J. (ed.) JWAC 2010 - 1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship. Saint Malo, France (2010)
3. Gupta, R., Tokekar, S.: Proficient pair of replacement algorithms on l1 and l2 cache for merge sort. J. OF COMPUTING 2(3), 171–175 (Mar 2010)

4. Gusev, M., Ristov, S.: Matrix multiplication performance analysis in virtualized shared memory multiprocessor. In: MIPRO, 2012 Proc. of the 35th International Convention, IEEE Conference Publications. pp. 264–269 (2012)
5. Gusev, M., Ristov, S.: Performance gains and drawbacks using set associative cache. Journal of Next Generation Information Technology (JNIT) 3(3), 87–98 (31 Aug 2012)
6. He, L., Sun, Y., Zhang, C.: Adaptive Subset Based Replacement Policy for High Performance Caching. In: Emer, J. (ed.) JWAC 2010 - 1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship. Saint Malo, France (2010)
7. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Fifth Edition: A Quantitative Approach (2012)
8. Intel: Intel smart cache (May 2012), `http://www.intel.com/content/www/us/en/architecture-and-technology/intel-smart-cache.html`
9. Ishii, Y., Inaba, M., Hiraki, K.: Cache Replacement Policy Using Map-based Adaptive Insertion. In: Emer, J. (ed.) JWAC 2010 - 1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship. Saint Malo, France (2010)
10. Jaleel, A., Theobald, K.B., Steely, Jr., S.C., Emer, J.: High performance cache replacement using re-reference interval prediction (rrip). SIGARCH Comput. Archit. News 38(3), 60–71 (Jun 2010)
11. Janapsatya, A., Ignjatović, A., Peddersen, J., Parameswaran, S.: Dueling clock: adaptive cache replacement policy based on the clock algorithm. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 920–925. DATE '10 (2010)
12. Lira, J., Molina, C., González, A.: Lru-pea: a smart replacement policy for non-uniform cache architectures on chip multiprocessors. In: Proceedings of the 2009 IEEE international conference on Computer design. pp. 275–281. ICCD'09, IEEE Press, Piscataway, NJ, USA (2009)
13. OpenMP: (2012), `https://computing.llnl.gov/tutorials/openMP/`
14. Pimple, M., Sathe, S.: Architecture aware programming on multi-core systems. International Journal of Advanced Computer Science and Applications (IJACSA) 2, 105–111 (2011)
15. Qureshi, M.K., Jaleel, A., Patt, Y.N., Steely, S.C., Emer, J.: Adaptive insertion policies for high performance caching. SIGARCH Comput. Archit. News 35(2), 381–391 (Jun 2007)
16. Reineke, J., Grund, D.: Relative competitive analysis of cache replacement policies. SIGPLAN Not. 43(7), 51–60 (Jun 2008)
17. Ristov, S., Gusev, M.: Achieving maximum performance for matrix multiplication using set associative cache. In: Computing Technology and Information Management (ICCM2012), 2012 The 8th Int. Conf. on. ICNIT '12, vol. 2, pp. 542–547 (2012)
18. Zhang, K., Wang, Z., Chen, Y., Zhu, H., Sun, X.H.: Pac-plru: A cache replacement policy to salvage discarded predictions from hardware prefetchers. In: Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. pp. 265–274. CCGRID '11, IEEE Computer Society, Washington, DC, USA (2011)