OPEN ACCESS

International

Journal

Of Modern Engineering Research (IJMER)

Effective Replacement Policy For Last-Level Cache

Hyun Kwon[1], Yongchul Kim[2,*]

[1]School of Computing, Korea Advanced Institute of Science and Technology, South Korea

[2]Department of Electrical Engineering, Korea Military Academy, South Korea

Corresponding Author: Hyun Kwon

ABSTRACT: *Since blocks stored in memory take a lot of time for the CPU to process, frequently used blocks are stored in the cache and placed between the CPU and memory. It is faster to access the cached blocks than the blocks stored in memory, hence the role of cache is important to improve the performance of computer systems. However, the cache has less capacity to store blocks than memory. Because of this, cache replacement policy, which replaces unnecessary blocks, is important. There are three cache replacement policies: random, least recently used (LRU), and most recently used (MRU). Generally, LRU method shows good performance. Cache is graded as Level 1 (L1), Level 2 (L2) and Level 3 (L3): L1 and L2 are usually referred to as higher level caches and L3 is referred to as last-level cache (LLC). The most frequently used blocks in the cache are stored in L1, while the less frequently used blocks are stored in L3. Therefore, the performance of LRU in L3 is limited. In this paper, we propose a method to improve Decision Tree Analysis in LLC. In the existing decision tree policy, some policy parts are modified to reduce storage space and we also reduce learning time and cache miss rate. Our experiment is implemented by using CMP$im and the performance evaluation uses SPEC CPU2000 and PARSEC 3.0 as the benchmark. Our results show that the proposed scheme improves the performance compared to the conventional LRU method with a minimal expense of cache miss. the cache are stored in L1, while the less frequently used blocks are stored in L3. Therefore, the performance of LRU in L3 is limited. In this paper, we propose a method to improve Decision Tree Analysis in LLC. In the existing decision tree policy, some policy parts are modified to reduce storage space and we also reduce learning time and cache miss rate. Our experiment is implemented by using CMP$im and the performance evaluation uses SPEC CPU2000 and PARSEC 3.0 as the benchmark. Our results show that the proposed scheme improves the performance compared to the conventional LRU method with a minimal expense of cache miss.*

KEYWORDS: *Last Level Cache (LLC), Least Recently Used (LRU), Most Recently Used (MRU), Decision Tree Analysis (DTA)*

I. INTRODUCTION

Due to the increasing gap between CPU and memory speed, cache performance plays a critical role in determining the overall performance of computer systems. Many cache performance models have been proposed in the literature. One of the key factors that affect cache performance is the cache replacement policy. Because the cache is limited in capacity, it is necessary to remove unnecessary blocks and replace them with new blocks. These replacement policies for caches have been studied for almost 5 decades [1][2]. Unlike offline replacement policy that relies on the knowledge of future address references, replacement policies that can be implemented in practice are online policies that have no knowledge of future references. Also, there does not exist an optimal replacement policy. All the replacement policies have the same average miss ratio due to the same cache's length, because the replacement policy consider the set of all the possible insertion sequences of a given length, with addresses taken from a finite set. Nevertheless, in terms of practical performances, not all replacement policies are equivalent. Algorithm-generated insertion sequences [1] have characteristics that make certain policies better than the others in practice. Among the replacement policies, the LRU policy is known to have the best performance. However, LRU's success is very dependent on a temporal locality. Last-level cache have less temporal locality than L1 and L2 caches. A large portion of the blocks stored in the last-level cache are never reused. Therefore, in the last-level cache, the LRU scheme has no significant effect and it is necessary to study an appropriate replacement policy for the last-level cache [3][4]. In this paper We propose a method to improve Decision Tree Analysis (DTA) in LLC.

In the existing decision tree policy, some policy parts are modified to reduce storage space and we also reduce learning time and cache miss. Our contributions are as follows:

- First, the goal of this paper is to implement well-known replacement policies in literature. Especially, we consider the case about Last-Level Cache. We modified decision tree presented in [5] and implement a hybrid-replacement policy in order to evaluate the performance.
- Second, we find an optimal policy by modifying some parts of the existing policies with a decision tree analysis. We found some limitations of the DTA in the existed works and provide better solution to improve the performances. The lower storage architecture is expected to eliminate useless branch in the decision tree.

This paper is organized as follows. In section 2, we have researched the related works regarding replacement policy. Then, our proposed scheme including policy implementation and source codes is presented in Section 3. Our experiment results are shown in Section 4 and we conclude in Section 5.

II. BACKGROUND AND RELATED WORKS

There are many various policies such as First In First Out (FIFO, Round-Robin) [6][7], Last In First Out (LIFO), Random [1], Least Recently Used (LRU) [2], Most Recently Used (MRU) [8], and PLRU (Pseudo-LRU) [9][10], Time aware Least Recently Used (TLRU) [11], Segmented LRU (SLRU) [12], Least-Frequently Used (LFU) [13], Least-Frequently Recently Used (LFRU) [14], Low Inter-Reference Recency Set (LIRS) [15], and Adaptive Replacement Cache (ARC) [16]. The FIFO method is a method of removing and inserting the first block accessed for the first time without a separate access history when the cache block needs to be replaced. Unlike FIFO, LIFO method is a method to insert and remove the last accessed block when replacement of cache block is needed. The Random method is a method of removing and inserting any existing block, i.e., randomly chosen block when it is necessary to replace the cache block. This approach has the advantage of not having to store the access history separately and will be useful when a stochastic method is effective. The LRU method removes and inserts the least recently used blocks. In order to apply this method, a separate history is needed to track how much the block is used in the history. Unlike the LRU method, the MRU method removes and inserts the most recently used blocks. This method is effective in situations where many old blocks stored in the cache are used for access. The PLRU approach improves LRU with less overhead and less cost than LRU for associative applications. This method is often used for CPU design because only one bit is used. The TLRU method is a modified LRU method used when a time element is inserted into a block stored in the cache. This TLRU method is widely used in network environments such as Information-centric networking (ICN) [17] and Content Delivery Network (CDNs) [18]. The SLRU method is a modified LRU method that distinguishes between probationary segments and protected segments. This method improves performance over LRU by storing more used blocks in the protected segment. The LFU method counts the number of blocks and removes and inserts the least number of blocks first. The LFRU method is a mixture of LRU and LFU methods. That is, replace the least frequently used and recently unused blocks. In this case, it is important to select appropriate weights for LFU and LRU in order to improve LFRU performance. The LIRS method is a method using an inter-reference recency (IRR) instead of the recently accessed history when the replacement policy is applied. The smallest IRR blocks will be removed and inserted. The ARC method is a method of combining recency and frequency and dynamically adjusting the recency and frequency with different weights according to importance. Among those cache replacement policies, it is not easy to determine the best performance policy. Al-zoubi et. al. [19] studied various cache replacement policies including above mentioned policies and showed comparison analysis using SPEC CPU2000 benchmark suite.

LRU is known as one of the efficient policy but has downside regarding storing cost. However LLC (last-level-cache) are not used efficiently in LRU policy. Because in order to access blocks stored in L3, L1 and L2 are checked first to see if there are corresponding blocks. Therefore, most recently used blocks by LRU policy of L1 and L2 are mostly stored in L1 and L2, and processed in advance by LRU policy in L1 and L2 before accessing L3. That is the most of the LLC blocks are rarely re-accessed by LRU policy due to filtering of the higher level (L1/L2) caches. As a solution to that problem, the authors in [4] present a bypassing method by analyzing the miss patterns of accessing or bypassing each cache block in the LLC, and predict the blocks that will not be re-accessed again in the LLC. However, cache bypassing algorithms suffer from several shortcomings. It needs a large storage overhead and a complex algorithm. Also, LRU has countless variations. Khan et. al. [5] introduce insert position select policy using Decision Tree Analysis. This technique requires minimal hardware modification from the novel LRU replacement policy. For a 1MB 16 way set-associative last level cache in a single core processor, this scheme uses only 2069 bits over the LRU replacement policy. The importance of this scheme is that users dynamically choose the insert position based on the decision tree. However, there are some limitations. The number of candidates are still limited because of the verification overhead of the decision tree analysis. These limits of the

insert position select policy reduce the chances to further improve energy efficiency of the dynamic cache mechanism.

III. PROPOSED SCHEME

We propose a new structure of the decision tree analysis to improve cache performance. Figure 1 shows a basic structure of decision tree. From the results of an existing DTA analysis, it can be seen that the benchmarks that measure near MRU and near LRU have little effect on the performance rating at 0% and 10%. Therefore, we remove the insertion parts near MRU and near LRU in order to reduce storage space and learning time as shown in Figure 2. Moreover, each decision position of the proposed scheme is generally reflected in the benchmarks performance evaluation.

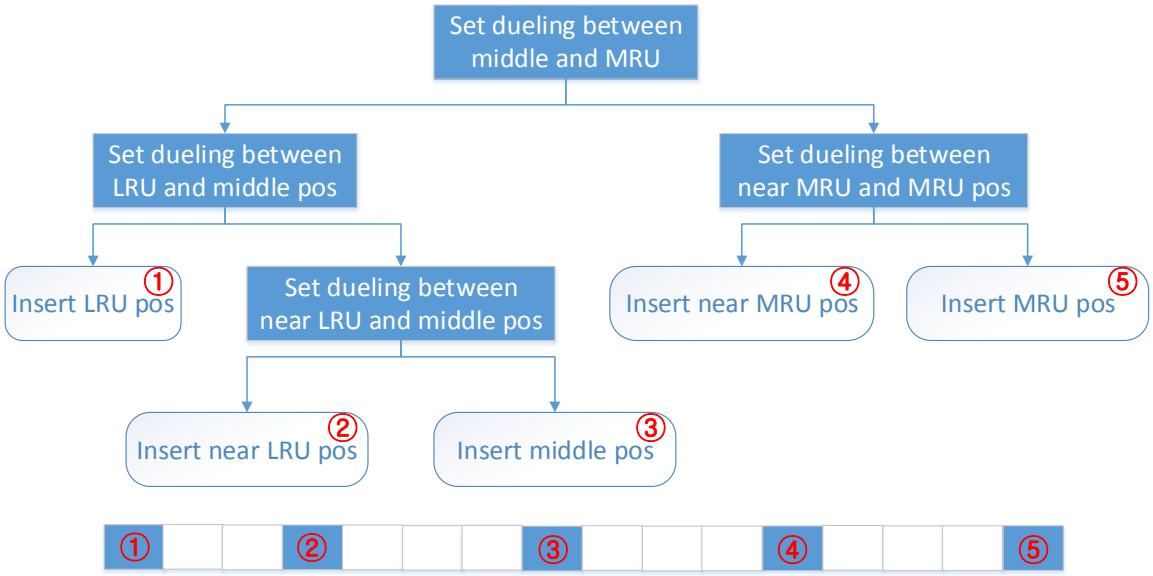

Fig. 1: A basic structure of the decision tree

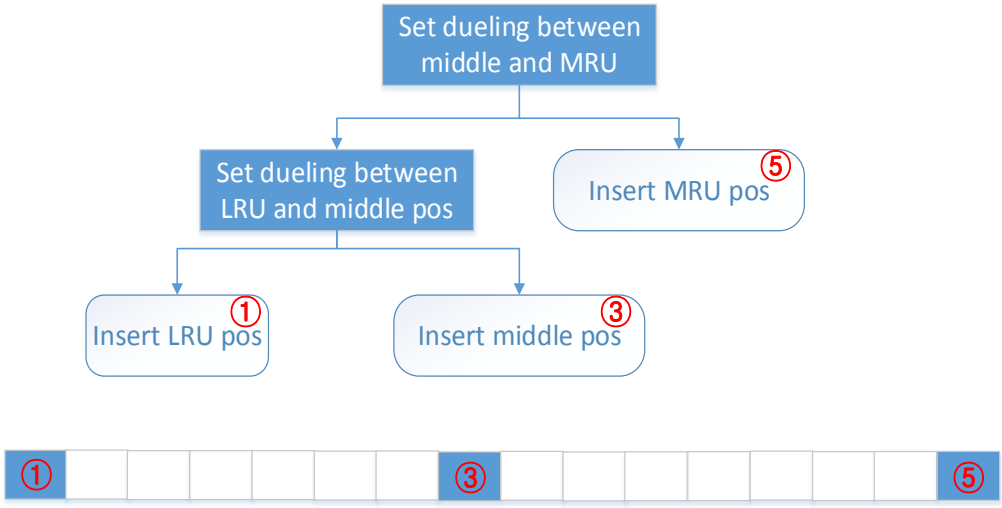

Fig. 2: A proposed structure of the decision tree policy

To implement the proposed method, we used CMP$im [20] as a simulator and SPEC CPU2000 was used to evaluate the performance of the proposed method. In addition, we evaluate the success and the fail performance of the proposed method using a multi-threaded program for the experiment of time performance using a streamcluster of PARSEC 3.0 [21].

We compared the proposed method with LRU and DTA for each cache miss rate experiment. A total of 50 million instructions were simulated. To implement the proposed method, we modified the replacement_policy.cpp part by using CMP$im as follows. This modified part can be expressed by an algorithm as shown in Algorithm 1.

<MRU>: count1 < numsets/2, count2 >= numsets/2
<Middle>: count1 >= numsets/2, switched = 0, count2 >= numsets/2
count1 >= numsets/2, switched = 1, count2 >= numsets/2
<LRU>: count1 >= numsets/2, switched = 0, count2 < numsets/2

Algorithm 1. Proposed replacement policy of cache.

Input: setType0 : Follower set
setType1 : Leader set that inserts in the middlePos.
setType2 : Leader set that inserts in the MRUPos, this is actually LRU policy.
setType3 : Leader set that inserts in a position adaptively chosen.
UINST32 *numsets*;
UINST32 *count1* : this counter is responsible for selecting winner policy of MRUPos.
UINST32 *count2* : this counter is responsible for selecting winner policy between 1st round And adaptive policy chosen by setType 3.
UINST32 *switched* : switched is a 1 bit count, it keeps track of the policy used in setType3
UINST32 *moiddel_Pos* : this is the insert position in the middle of the LRU stack

Process of proposed scheme:
If (count1 <numset/2) **Then** // MRU position insertion is winner in 1st round
InsertToPos(new_block); // middelPos insertion is winner in 1st round
Else
If (switched == 0) **Then**
If (count2 >= numsets/2) **Then**
InsertToPos(new_block) // middle position insertion is winner in 2nd round
Else
InsertToPos(new_block) // LRU position insertion is winner in 2nd round
Else if (switched == 1) **Then**
InsertToPos(new_block) // MRU position insertion is winner in 2nd round
End If

Fig. 3: Experimental step

In order to evaluate the performance of the proposed method, we experimented with the steps of creating trace files and simulating them as shown in Figure 3. To generate the trace files, four trace files such as art, mcf, twolf, parser are created on the SPEC2000 benchmark using Pin (Pinkit) and a streamcluster is created in PARSEC3.0. And then, we use the trace files to evaluate the cache performance of the proposed method implemented with CMP$im.

IV. SIMULATION RESULTS

In order to compare the performance of the proposed method, performance was evaluated using CMP$im. Figure 4 shows the performance evaluation using the twolf benchmark in CMP$im simulator. Performance of the cache miss rate was measured using four art, mcf, twolf, and parser as benchmarks for performance evaluation. Cache miss means that the CPU does not have the memory to be referred to in the cache. When there are many cache misses, it takes a long time to fetch the information because the data of a certain block site is fetched from the main memory, stored in the cache, and then transmitted to the CPU. The total number of cycles was 50 million cycles.

```
[CMP$im] Finished Requested Instruction Count 50000001  for App 3 at Time: 50715094
[CMP$im] Finished Requested Instruction Count 50000001  for App 1 at Time: 50724893
[CMP$im] Finished Requested Instruction Count 50000001  for App 0 at Time: 50777410
[CMP$im] Finished Requested Instruction Count 50000001  for App 2 at Time: 50911240


Full Run Summary: (Global Instructions: 200546036)
        Thread ID: 0 ICOUNT: 50145464 CYC: 50911241 CPI: 1.01527 Global Cycles: 50911241 LLC Misses: 262372
        Thread ID: 1 ICOUNT: 50195245 CYC: 50911241 CPI: 1.01426 Global Cycles: 50911241 LLC Misses: 262381
        Thread ID: 2 ICOUNT: 50000001 CYC: 50911241 CPI: 1.01822 Global Cycles: 50911241 LLC Misses: 262553
        Thread ID: 3 ICOUNT: 50205326 CYC: 50911241 CPI: 1.01406 Global Cycles: 50911241 LLC Misses: 262530


Region of Interest Summary:
        Thread ID: 0 ICOUNT: 50000001 CYC: 50777410 CPI: 1.01555 Global Cycles: 50911241 LLC Misses: 261720
        Thread ID: 1 ICOUNT: 50000001 CYC: 50724893 CPI: 1.0145 Global Cycles: 50911241 LLC Misses: 261461
        Thread ID: 2 ICOUNT: 50000001 CYC: 50911240 CPI: 1.01822 Global Cycles: 50911241 LLC Misses: 262553
        Thread ID: 3 ICOUNT: 50000001 CYC: 50715094 CPI: 1.0143 Global Cycles: 50911241 LLC Misses: 261573
```

Fig. 4: An example of the execution result about twolf using CMP$im.

Table. 1: Cache miss cycle (unit: cycle) of LRU, decision tree, and proposed solution.

	LRU				DECISION TREE				OUR SOLUTION			
	T1	T2	T3	T4	T1	T2	T3	T4	T1	T2	T3	T4
art	128167	129213	120600	129650	69757	68295	65672	71567	84123	89344	86595	89628
mcf	809865	809740	809937	809598	639165	637560	641446	642711	756656	755026	755340	755611
twolf	311239	311190	311171	311077	261995	260516	260281	260964	261720	261461	262553	261573
parser	93248	93123	93073	93159	93666	93639	93693	93736	93766	93725	93794	93847

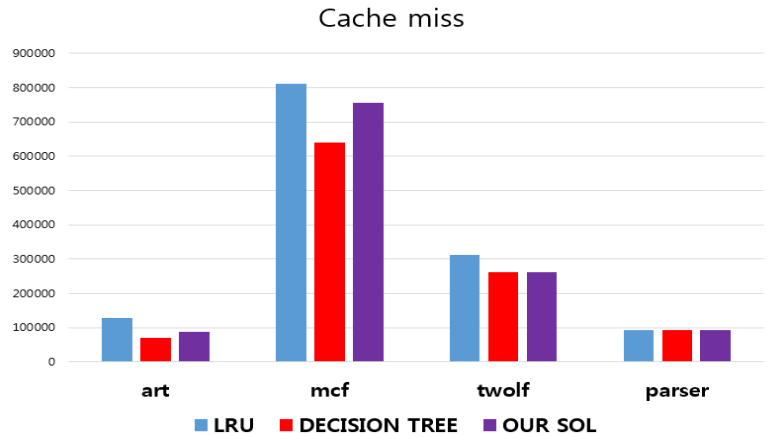

Fig. 5: Graph of Table. 1

Table 1 shows lists the cache miss cycle of itemized LRU, DTA, and proposed scheme. Figure 5 is a graphical representation of the results of Table 1. In Table 1 and Figure 5, we can see that the overall decision tree has a lower cache miss rate than the proposed solution. However, in the twelf and parser parts, we found that the cache miss rate is similar to the decision tree and the proposed solution. Overall, Both the proposed method and the DTA method showed less cache miss rate than the existing LRU method. The goal of the proposed solution is to improve running time and storage space. We need comparative analysis between DTA and our solution, but we didn't find solution about the method of comparison.

V.CONCLUSION

The last-level cache mitigates the impact of long memory latencies in today's microarchitectures. The replacement policy in the LLC can have a significant impact on cache efficiency. However, a fixed replacement policy can allow useless blocks to remain in the cache longer than necessary, resulting in inefficiency. The selection of replacement policy using decision tree analysis of multi-set dueling is a simple efficient technique that can be implemented in hardware with minimal change and minimal additional hardware cost. In this paper, we propose a new method to improve Decision Tree Analysis in LLC by modifying selection policy that eliminates useless decision branch. Our simulation results show that the proposed scheme achieves better performances regarding running time and storage space.

REFERENCES

[1]. M Ozaki, Y. Adachi, Y. Iwahori, and N. Ishii, Application of fuzzy theory to writer recognition of Chinese characters, *International Journal of Modelling and Simulation, 18(2)*, 1998, 112-116. (9)

[2]. Zhou, S. (2010, September). An Efficient Simulation Algorithm for Cache of Random Replacement Policy. In NPC (pp. 144-154).

[3]. Jelenković, Predrag R., and Ana Radovanović. "Least-recently-used caching with dependent requests." Theoretical computer science 326.1 (2004): 293-327.

[4]. Albericio, Jorge, et al. "The reuse cache: downsizing the shared last-level cache." Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2013.

[5]. J, Gaur, M. Chauduri., et al. Bypass and Insertion Algorithms for Exclusive Last-Level Caches. In: Proc. 38th int. Symposium on Computer Architecture (ISCA), 2011. Cache replacement

[6]. KHAN, Samira; JIMÉNEZ, Daniel A. Insertion policy selection using decision tree analysis. In: Computer Design (ICCD), 2010 IEEE International Conference on. IEEE, 2010. p. 106-111.

[7]. Guan, Nan, et al. "FIFO cache analysis for WCET estimation: A quantitative approach." Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium, 2013.

[8]. Joyce, Thomas F. "Round robin replacement for a cache store." U.S. Patent No. 4,195,343. 25 Mar. 1980

[9]. Mounes-Toussi, Farnaz. "Method and apparatus for miss sequence cache block replacement utilizing a most recently used state." U.S. Patent No. 6,098,152. 1 Aug. 2000.

[10]. Smith, Michael B., and Michael J. Tresidder. "Pseudo-LRU cache memory replacement method and apparatus utilizing nodes." U.S. Patent No. 5,594,886. 14 Jan. 1997.

[11]. Grund, Daniel, and Jan Reineke. "Toward precise PLRU cache analysis." OASIcs-OpenAccess Series in Informatics. Vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[12]. Bilal, Muhammad, and Shin-Gak Kang. "Time aware least recent used (TLRU) cache management policy in ICN." Advanced Communication Technology (ICACT), 2014 16th International Conference on. IEEE, 2014.

[13]. Gao, Hongliang, and Chris Wilkerson. "A dueling segmented LRU replacement algorithm with adaptive bypassing." JWAC 2010-1st JILP Worshop on Computer Architecture Competitions: cache replacement Championship. 2010.

[14]. Lee, Donghee, et al. "On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies." ACM SIGMETRICS Performance Evaluation Review. Vol. 27. No. 1. ACM, 1999.

[15]. Lee, Donghee, et al. "On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies." ACM SIGMETRICS Performance Evaluation Review. Vol. 27. No. 1. ACM, 1999.

[16]. Jiang, Song, and Xiaodong Zhang. "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance." ACM SIGMETRICS Performance Evaluation Review 30.1 (2002): 31-42.

[17]. N. Megiddo and D.S. Modha, "Arc: A Self-Tuning, Low Overhead Replacement Cache," Proc. Second USENIX Conf. File and Storage Technologies, 2003.

[18]. Ahlgren, Bengt, et al. "A survey of information-centric networking." IEEE Communications Magazine 50.7 (2012).

[19]. Choi, Jaeyoung, et al. "A survey on content-oriented networking for efficient content delivery." IEEE Communications Magazine 49.3 (2011).

[20]. AL-ZOUBI, Hussein; MILENKOVIC, Aleksandar; MILENKOVIC, Milena. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In: Proceedings of the 42nd annual Southeast regional conference. ACM, 2004. p. 267-272.

[21]. Jaleel, Aamer, et al. "CMP$im: A Pin-based on-the-fly multi-core cache simulator." Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA. 2008.

[22]. Chasapis, Dimitrios, et al. "PARSECSs: Evaluating the impact of task parallelism in the PARSEC benchmark suite." ACM Transactions on Architecture and Code Optimization (TACO) 12.4 (2016): 41.

Hyun Kwon." Effective Replacement Policy For Last-Level Cache" International Journal Of Modern Engineering Research (IJMER), vol. 08, no. 01, 2018, pp. 01–06.