In Situ Caching using Combined TTL-FIFO Algorithm

Gasydech Lergchinnaboot
Department of Mathematics and Computer Science organization
Chulalongkorn University
Bangkok, Thailand
e-mail: Gasydech@gmail.com

Peraphon Sophatsathit
Department of Mathematics and Computer Science organization
Chulalongkorn University
Bangkok, Thailand
e-mail: Peraphon.s@chula.ac.th

Saranya Maneeroj
Department of Mathematics and Computer Science organization
Chulalongkorn University
Bangkok, Thailand
e-mail: Saranya.m @chula.ac.th

***Abstract*— This research proposes an algorithmic cache arrangement scheme to efficiently utilize existing hardware that are currently plagued with memory wall problem. The proposed scheme exploits straightforwardness of First-in, First-out (FIFO) scheduling algorithm and in situ placement technique. FIFO allows the proposed scheme a fair caching of processes. In situ replacement economically utilizes spaces by replacing the expired process with a new process in the same memory space without flushing. This combination helps reduce operating overheads, which in turn lower power consumption. The benefits of their simplicity and hardware implementable will accelerate operational speed that eventually closes the gap between processing speed and memory access/retrieval speed, thereby lessens the memory wall problem.**

Keywords-Algorithmic arrangement; Cache; FIFO; Simulation.

I. INTRODUCTION

Computers have been extensively used in many facets of life for decades. Early in their lifetime, the numbers of applications were limited and used by specialists. The enormity of computer size was perhaps their trademark. As technology progresses, their dimensions change drastically, ranging from small embedded devices, handheld gadgets to cabinet size supercomputers. Regardless of sizes, they do share one similarity system architecture. Most systems still contain 4 main components, namely, input, processing, memory, and output units.

This research focuses on two components, i.e., processor and memory. These two components are evolved at different paces. The processing unit is pushing 50 percent per year, while the memory unit is merely crawling at 7 percent[1]–[4]. Such progress discrepancies widen their mutual computation capabilities that eventually lead to the infamous memory wall problem. This problem will cause performance bottleneck which can diminish system throughput considerably.

A system is said to be superior if these bottlenecks are either reduced or eradicated. However, in practice bottlenecks cannot be totally obliterated. A number of researches have addressed these issues by either using more advanced hardware, reducing data that pass through system's critical modules, or introducing efficient algorithms. Unfortunately, these attempts possess a couple of technical viewpoints. First, using hardware to improve or fix the bottleneck problems with advanced hardware is definitely not a long-lasting solution. Second, by introducing simple and straightforward memory replacement algorithms that can virtually operate on existing hardware, the operating overhead that incurred by all memory related software systems can be procedurally reduced. Hence, the memory wall problem is, to an extent, alleviated.

The proposed approach will exploit this latter viewpoint by re-arranging memory structure to permit fast access, simple organization, and low operating overhead. These goals will assume the following conditions that have adverse effects on existing memory management scheme:

- Increasing number of core counts.
- Rising CPU clock speed.
- Increasing cache size.
- Increasing number of cache blocks.

A proposed algorithm will rearrange system caching order using the simplest First-In First-out (FIFO) allocation scheme. New execution contents are then loaded in situ. In so doing, no cache manager is needed to add on any overhead. This will make it distinguishable from existing approaches. Details on how the proposed approach is laid out and operated will be described in the sections that follow.

This paper is organized as follows. Section 2 summarizes some related works that are essential to the construction of the proposed approach. Section 3 describes the work being performed in detail. Section 4 explains the simulation process and evaluation of the proposed scheme in comparison with existing cache systems. Discussion on further technical aspects is given in Section 5. Future

978-1-7281-4742-0/19/$31.00 ©2019 IEEE

potential of the proposal scheme and enhancement are given in the last section.

II. Related Works

While design and implementation of processing units advance for decades, memory units are gradually improving. Flash storage was introduced to speed up storage and retrieval operations. Unfortunately, memory management process was still incapable of exploiting faster hardware capability. Wu and Kuo [5] addressed this issue by introducing 2 types of translation layer to memory management system, i.e., Flash translation layer (FTL) and NAND Flash translation layer (NFTL). These translation layers have unique characteristics. FTL uses page-level address translation mechanism, which takes shorter time to translate addresses, less space, and has considerably low garbage collection overhead. On the other hand, NFTL adopts block-level address translation mechanism which has low memory requirement to operate. By combining these two mechanisms, we can exploit and dynamically switch between 2 translation mechanisms to maximize performance and optimize space utilization.

Parallel file systems (PFS) have become a choice to minimize performance throttling due to I/O bottlenecks in computer systems. However, traditional storage drives and small requests were still able to lessen the overall system performance problems. He, Wang, and Sun [6] introduced a Selective and Layout-Aware Cache system that exploited small set of solid-state drives (SSD). The most distinguishable characteristic of SSD is minimal latency, small power consumption, and high bandwidth. Nevertheless, the operating costs of SSD are relatively high. Proper algorithms need to be introduced to limit the number of flash storage usage. At any rate, when systems are less crowded, every process will be assigned to high performance cache to maximize system performance.

There are two levels of memory hierarchy for memory page management. The first level or cache is the closet to CPU having smaller storage capacity to fit inside CPU for fast access time. The second layer or main memory is farther from CPU having larger capacity but takes considerably longer access time. Chrobak and Noga [7] proposed a page fault removal algorithm to minimize operating time, cost, and maximize system performance. To achieve these goals, only valid data (under processing) are allowed in cache and invalid data (obsolete) are eliminated. Eviction decisions are made without knowledge of future memory request. Since all data transfers must be performed as fast as possible to keep pace with the CPU speed, the simplest and most efficient method is used. Typical SSD caches employ Least Recently Used (LRU) algorithm. However, one crucial consideration is page fault handling. FIFO removes the longest resident page, whereas LRU removes the longest unused page. Without reiterating their pros and cons, in particular, starvation which was taken care of by TTL, we opted for FIFO algorithm due to its simplicity, virtually required no overhead as it was directly hardware implementable [8], which is in contrast to the study by Chrobak and Noga [7]. The concerns on how much resources have been utilized to weigh the costing tradeoffs are handled at the operating system level.

Gomaa, Messier, and Davies [9] employed cache consistency mechanism to ensure cache validity in system. They utilized two variants of weak consistency, typical Time-to-Live (TTL-T) and Time-to-Live immediate ejection (TTL-IE). The TTL employment system requires less overall bandwidth than conventional cache system by renewing valid cache blocks and evicting invalid cache blocks according to variation of TTLs. However, the performance of the proposed system heavily depends on the data expiry rate.

Berger, Henningsen, Ciucu, and Schmitt [10] also utilized Time-to-Live (TTL) policy to increase hit/miss ratio in cache system. Utilization was determined by having multiple decay rates. For a hit page, it will be renewed with extra TTL. While less popular pages, their TTL will be reduced to accelerate decay rate. The introduction of TTL variation was able to increase the hit ratio from 0.64 to 0.77.

III. Proposed Method

This section explains how the simulation is organized in this experiment, covering memory structure, prior problems, their root causes, and possible improvement. Chosen choices, design rationale, and system architecture will be thoroughly described in this section.

Currently, consumer grade computer system contains 3 main components, namely, cache, main memory, and auxiliary memory. According to Rixner's work, these memory systems were pushing 7 percent of performance [11]. Consequently, structural adjustments have been made, namely, increasing number of cache blocks, widening bandwidth capability, and expanding storage capacity.

These advancements have yet outperformed the processing performance owing to their performance gap over the years. Advanced hardware will not lessen the bottleneck problem, thereby the root cause still persists.

A. System Architecture

A proposed a new algorithmic cache structure arrangement aims to minimize the operating overhead, yet be able to yield a comparable result to existing systems. Implementation details are described in 3 parts, i.e., system architecture, input generator, and executor.

The proposed system architecture consists of 3 main components, namely, process distributor, memory pool, and result accumulator, as shown in Figure. 1.

Figure 1. System architecture.

The above scheme is set up in a black-box configuration to offer maximal transparency to system implementor by making the following assumptions:

1. The process distributor represents some operating system modules that handle process management.

Common working modules are scheduler, dispatcher, etc.

2. The memory pool is the focus of this research that will be explained subsequently.
3. The result accumulator represents additional modules or functionalities run by the operating system.

Assumptions 1) and 3) are not part of this research responsibility. We design 2) as a plug-in module that can be installed on any compatible hardware configurations. This pool will utilize as much the underlying hardware capabilities as possible, e.g., FTL and NFTL, to minimize re-inventing the wheel. Yet the emphasis lies on TTL-FIFO in situ caching algorithm. This implementation follows Intel® cache structure. In current CPU setting, there are 3 layers, namely, L1, L2 and L3. The closer cache to processing cores, the smaller the size and faster transfer speed. Each processing core is assigned its own set of cache blocks, except L3 which is a common shared cache among cores as shown in Figure. 2.

Process distributor manages incoming tasks by splitting each task into processes and distributing them over available cache blocks. The process distributor breaks a sizable task into smaller processes that fit the L3 cache. If L3 is full, the process distributor will hold the newly split processes until space is available.

Process allocation will be carried out based on individual process arrival time. As processes arrive in L3, allocation order commences using FIFO scheduling algorithm. All processes waiting in L3 will subsequently be moved up to L2 and L1 and get executed by the CPU.

When the CPU needs to read, it looks for the desired page through cache one level at a time. If the desired page is in L1, data will be marks as a hit. Otherwise, it is a miss. A miss triggers a search in the next level, or L2, and so on. If the page is not in cache, the main memory will be searched next until the page is found. Otherwise, an operating system page fault is triggered to initiate subsequent disk I/O read-in.

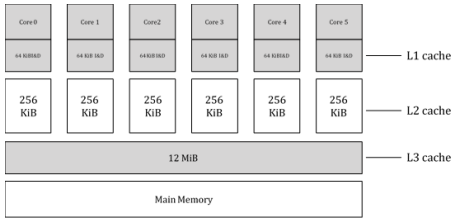

Figure 2. Intel® Coffee lake cache organization.

There are two types of process ranking, namely, global and local. Global ranking system signifies the process class, which is primarily system processes. Local ranking, on the other hand, denotes common user processes. A numerical value ranges from 0 to 255 is set up to represent process ranking, where lower value process has higher ranking to be allocated first in the memory pool.

When a system process arrives in memory pool, there are 3 situations to be determined:

1. Memory pool is free.
2. Memory pool is fully occupied by user processes.
3. Memory pool is fully occupied by system processes.

If the memory pool is free, the system will simply push the first process in FIFO queue to next available memory slot. If the memory pool is fully utilized and a high system process arrives, some user processes must be removed. Removing process from working area is done by removing the latest arrival process since it unlikely uses any resources. Finally, if the system processes fully occupy the memory pool, the system will halt from being overloaded.

The last component is result accumulator. This component simply collects and reassembles the resulting data on the next available space. This data gathering process serves as a verification for completion status to be notified the operating system.

The proposed architecture is designed, for all practical purposes, to be independent of the underlying hardware, as long as the cache arrangement is compatible.

B. Implementation

When a process is dispatched by the process distributor to the memory pool, the memory pool assigns a global ranking value and stores it in L3. Eventually, all the processes will be promoted to L1 for immediate execution. This is depicted in Figure. 3(a). Notice that user processes are ranked above 128 (in gray). When a process completes its execution, it is dispatched to the result accumulator to be handed over to the operating systems. However, some processes have not yet completed the execution and their time-slice (TTL) is not expired. They will be automatically renewed their L1 residency duration, wherein the processing status will remain active (or 1), as depicted in Figure. 3(b). This matrix is used by the execution loader to replace L1 in situ at the 0-mask positions and leave the 1-mask positions unreplaced. This procedure repeats forever to keep L1, L2, and L3 occupied for feeding the processor unit. As such, considerable memory transfer overheads are reduced. The procedure is summarized in Algorithm 1 as shown below.

2	26	53	71
14	76	0	82
7	47	93	143
181	156	221	91

(a)

0	0	0	0
0	0	0	0
0	1	0	0
0	0	1	0

(b)

Figure 3. Replacement mask

(a) process rank number in cache block (b) processing status.

Algorithm1: (Process Distributor – Memory Pool – Result Accumulator)

Input: pid of P, attribute (Process id, user/system)

Output: results of P

Begin

allocate $P_i \in P$ on L3

identify type of P_i // System or User

assign process value // 0-127 or 128-255
fill $P_i \in P$ to L2 // Until L2 is full
find finished process P_j // P_j is in L1
mask all finished processes //Set 0-mask
move to RA // Result Accumulator
renew unfinished processes // extend TTL, set 1-mask
replace P_j by Pi in situ // From L2 to L1
End

IV. EXPERIMENTAL RESULTS

This section discusses the proposed method outcomes, covering experimental set up, input sequences, and proposed method performance.

A. Experimental setup

The experiment was simulated on Ubuntu 16.04.6 LTS, using Intel® i7-8700 with 12 MB cache, Corsair® DDR4 2666 MHz configured as 2x8 GB main memory. The instructions were stored in M.2 NVMe drive at read/write speed of 3200/2000 Mbps. Other parameters such as page size, data transfer rate, etc., were set to the chipset defaults. One important assumption was imposed on L1 to keep data on the entire L1 for execution, whereas the actual L1 stored both data and instructions. Coding was written in C since it could interact directly with the memory unit.

B. Input sequences

The input parameter values as shown in Figure.4 were randomly generated to prevent biasness.

Sign	Priority	ProcID	SubProc	SubID	Time	Value
Int	Int	Int	Bool	Int[]	Int	Int

Figure. 4. Input sequence structure.

Each field describes the following definitions:

- Sign bit indicates process type, where 1 denotes system process and 0 as user process.
- Priority shows global ranking of each process.
- ProcID shows number system used to track processes passing through a system.
- SubProc marks the relationship among processes.
- SubID contains list of all related processes. Positive values identifies as parent process and negative values as child process.
- Time indicates minimum time required to finish execution.
- Value stores actual value of individual process.

C. Performance evaluation

In this experiment, 100,000 processes were executed to evaluate effectiveness of the overall system. The experiment was run on traditional cache system using Least Recently Used (LRU) scheme and the proposed scheme. The following result statistics were collected: average number of accepted processes, average number of rejected processes (owing to the unavailability of cache), execution time, and number of processes removed from memory pool by LRU. This removal statistic did not exist in the proposed method by virtue of replacement in situ scheme.

From the total 100,00 processes, no process was rejected in the proposed method, as oppose to 3,406 in LRU cache system. The proposed system outperformed LRU cache system in terms of execution time. The average size of input sequence took 77 clock ticks and overall used 8,363,020 clock ticks to finish executing 100,000 processes. The LRU cache system took 9,293,416 clock ticks to accomplish the same task as shown in Figure. 5 and Figure. 6.

TABLE I. THE ARRANGEMENT OF CHANNELS

Scheme	Accepted process	Rejected Process	Execution time (per 100,000)	Removal
Proposed system	100,000	0	8,363,020	0
LRU cache system	96,594	3,406	9,293,416	100,000

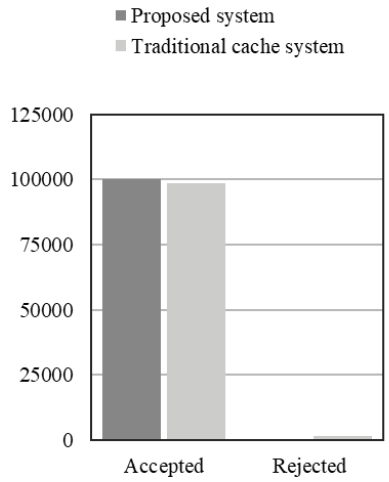

Figure 5. Accepted, rejected processes

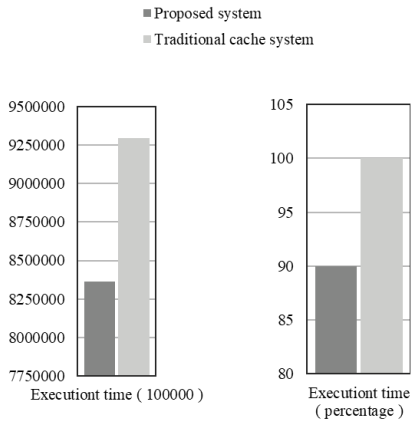

Figure 6. Time to accomplish 100,000 processes

The power consumption can be calculated by the following equation:

$X1 = [(accept + reject + remove) * t + execution] * P$ (1)

where P denotes power consumption per CPU cycle [3], which is equal to $5.7778 * 10^{-11}$, t denotes number of transfer movements. Table 2 shows the power consumption calculation, given CPU is running at 3.2 GHz and thermal design power (TDP) is 65 watts. Accept, reject, and execution operations take 3 cycles to operate. Substituting into (1), we have

- proposed system = $[(100{,}000 + 0 + 0) * 3 + 8{,}363{,}020] * 5.7778 * 10^{-11}$
- LRU cache system = $[(96{,}594 + 3{,}406 + 100{,}000) * 3 + 9{,}293{,}416] * 5.7778 * 10^{-11}$

TABLE II. POWER CONSUMPTION STATISTICS

Scheme	Power consumption	Difference
Proposed system	$5.0053 * 10^{-4}$	0%
LRU cache system	$5.7162 * 10^{-4}$	+14.2029%

which means the LRU system consumes more power than the proposed system by 14%. The cost of total caching is determined by the following equation:

$$C = \text{rejection} + \text{removal} + \text{execution}$$
$$= (r * t1 * A) + (f * t2 * B) + (e * E + X1) \quad (2)$$

where r, f, e, t_i denote the number of rejected processes, flushing processes, execution duration (in clock ticks), and clock tick for i = rejection (1), removal (2), respectively, and A, B, E denote the cost of resubmission, transfer, and execution per clock tick, respectively. Substituting into (2), we have

proposed system = $(0) + (0) + (8{,}363{,}020 * 5.0053 * 10^{-4})$
LRU cache system = $(3406 * 2.8889 * 10^{-10}) + (100000 * 1.7333 * 10^{-10}) + (9{,}293{,}416 * 5.7162 * 10^{-4})$

Table 3 shows the resulting estimates, assuming $A = 2.8889 * 10^{-10}$, $F = 1.7333 * 10^{-10}$, $E = 1.7333 * 10^{-10}$, and $P = 5.7778 * 10^{-11}$. Hence, the cost to operate the proposed system is reduced by 21% from the current LRU system.

TABLE III. POWER CONSUMPTION STATISTICS

Scheme	Cost	Percentage reduction
Proposed system	$4.1860 * 10^{3}$	78.6650%
LRU cache system	$5.3213 * 10^{3}$	100%

V. DISCUSSION

This paper exploited the simplicity of FIFO scheduling and in situ replacement scheme to improve caching efficiency. Processes coming from Process Distributor underwent normal cache promotion, i.e., from L3-L2-L1. The focus of this work was to arrange this promotion in the fastest and most efficient manner. We devised the replacement in situ scheme to reduce memory flushing load. Consequently, less overhead was incurred in cache transmissions and reallocations. The operating cost was 21.34% less than the existing LRU implementation.

To preserve the work-in-progress contents, we employed write-through scheme to keep the most up-to-date copy of data. Cache and memory utilization would run faster to keep pace with the CPU speed. Thus, the proposed scheme would co-exist and run with the underlying hardware smoothly.

VI. CONCLUSION

In this paper, the proposed scheme utilized existing cache system by altering cache arrangement algorithm and introducing minimal replacement procedure. By exercising FIFO algorithm and in situ replacement allowed the proposed scheme to provide comparable performance using less resources to achieve identical jobs. Since replacement activities were reduced, the proposed system required few parameters to run, thereby system overhead was also decreased.

Perhaps the most important contributions of this work are its low overhead and simplicity which render the proposed scheme to be hardware deployable. This would definitely not duplicating or reinventing the wheel of any operating system functions. Future work will be focused on placement algorithms to better utilize cache space and lower power consumption as the degree of multiprocessing increases. It is envisioned that such an undertaking will be conducive toward alleviating the memory wall problem.

REFERENCES

[1] E. P. Debenedictis, "It's Time to Redefine Moore's Law Again," Computer (Long. Beach. Calif)., vol. 50, no. 2, pp. 72–75, 2017.

[2] G. Strawn and C. Strawn, "Moore's Law at Fifty," IT Prof., vol. 17, no. 6, pp. 69–72, 2015.

[3] D. A. P. John L. Hennessy, "Computer Architecture: A Quantitative Approach."

[4] L. B. Kish, "End of Moore ' s law : thermal (noise) death of integration in micro and nano electronics," Phys. Lett. A, vol. 305, pp. 144–149, 2002.

[5] C. H. Wu and T. W. Kuo, "An adaptive two-level management for the flash translation layer in embedded systems," IEEE/ACM Int. Conf. Comput. Des. Dig. Tech. Pap. ICCAD, pp. 601–606, 2006.

[6] S. He, Y. Wang, and X. H. Sun, "Improving Performance of Parallel I/O Systems through Selective and Layout-Aware SSD Cache," IEEE Trans. Parallel Distrib. Syst., vol. 27, no. 10, pp. 2940–2952, 2016.

[7] M. Chrobak and J. Noga, "LRU is Better than FIFO," Algorithmica, pp. 180–185, 1999.

[8] G. Lergchinnaboot and P. Sophatsathit, "A biological-like memory allocation scheme using simulation," Proc. - 2017 2nd Int. Conf. Inf. Technol. Inf. Syst. Electr. Eng. ICITISEE 2017, vol. 2018-Janua, pp. 426–429, 2018.

[9] H. Gomaa, G. G. Messier, and R. Davies, "Hierarchical Cache Performance Analysis under TTL-Based Consistency," IEEE/ACM Trans. Netw., vol. 23, no. 4, pp. 1190–1201, 2015.

[10] D. S. Berger, S. Henningsen, F. Ciucu, and J. B. Schmitt, "Maximizing Cache Hit Ratios by Variance Reduction," ACM SIGMETRICS Perform. Eval. Rev., vol. 43, no. 2, pp. 57–59, 2015.

[11] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," Proc. 27th Int. Symp. Comput. Archit. (IEEE Cat. No.RS00201), vol. 27, no. c, pp. 1–11, 2000.