A Page Replacement Algorithm Based on Frequency Derived From Reference History

Hong-Bin Tsai
Department of Electrical Engineering
National Taiwan University, Taipei, Taiwan
hbtsai@ntu.edu.tw

Chin-Laung Lei
Department of Electrical Engineering
National Taiwan University, Taipei, Taiwan
cllei@ntu.edu.tw

ABSTRACT

Page replacement algorithm is one of the core components in modern operating systems. It decides which victim page to evict from main memory by analyzing attributes of pages referenced. The evicted page is then moved to backing store in the memory hierarchy, and moved back to main memory once referenced again. The technique that utilizes storage as part of memory is called swapping. However, there is a non-trivial performance gap between memory and storage. For example, performance of permanent storage like Solid-State Disk (SSD) is much slower, e.g. 10^4 longer write latency, than DRAM [9]. As a result, swapping between main memory and storage causes system performance to a discernible drop. Nevertheless, a higher hit ratio of page replacement algorithm implies less I/O waits to storage, and consequently a better performance overall.

In this paper we propose a log-based page replacement algorithm that assumes better hints for page replacement can be approached through analysis of page reference history. The algorithm selects victim page that holds lowest reference rate in a window-sized log. A simulation shows that our method outperforms conventional page replacement algorithms by 11% at best.

CCS Concepts

•Software and its engineering → Operating systems; Memory management; Virtual memory;

Keywords

page replacement algorithm; virtual memory management

1. INTRODUCTION

Memory-intensive and data-intensive services, e.g, big-data analysis [3], are profoundly developed in modern years. The demand for larger memory and faster CPU increases simultaneously. However, restricted by energy consumption and density, the fabrication of DRAM technology has come to a scaling limit [10]. Consequently, program data and file data are inevitable to move between memory and disk when necessary. Performance is thus declined by waiting for completion of storage accesses. In other words, reducing data movements to storage improves system performance. Therefore a virtual memory management that minimizes page faults and storage I/O waits is keen to system performance.

Virtualizing memory addresses enables a system to use memory exceeding its hardware limit by swapping. However, the down side is performance loss in each swap operation due to the native performance gap between storage and memory. Considering the expensive cost of memory hardware and its technical limit in scale-up, it is impractical to build a memory as large as in data scale. It is also observed that data volume and complexity of application software are expanding rapidly [5], swapping is inevitable and in urgent need for improvements.

Storage technology has evolved greatly in recent years, and the penalty of swapping declines simultaneously. The cutting-edge persistent memory featuring byte-addressability and non-volatility emerges as a new tier in memory hierarchy. It shares similar access speed as main memory, and yet, with larger capacity [7]. We have conducted an experiment on CentOS 7 to test the performance of swap on persistent memory and found out a software bottleneck still exists.

1.1 Motivation

Restricted by limited availability of persistent memory, we have enabled a persistent memory emulator supported in Linux 4.5 [1] and configured it as a swap disk. A `qsort` utility from `stress-ng`[2] is run with various number of workers in parallel sorting 4,194,304 random integers. Its elapsed time for each worker to finish sorting is evaluated as shown in Figure 1. For comparison, we have run the same test on a ram-only configuration as well.

The persistent memory emulator is ram-based, implying that there is no hardware-induced performance gap between memory and swap space. Therefore, the I/O waits of moving data between memory and swap space are identical. We have observed that when the number of workers is increased, i.e., the more memory space is allocated and swapped, the gap of elapsed time between `qsort` run on ram-only and pmem-

SAC'17, April 3-7, 2017, Marrakesh, Morocco

http://dx.doi.org/10.1145/3019612.3019737

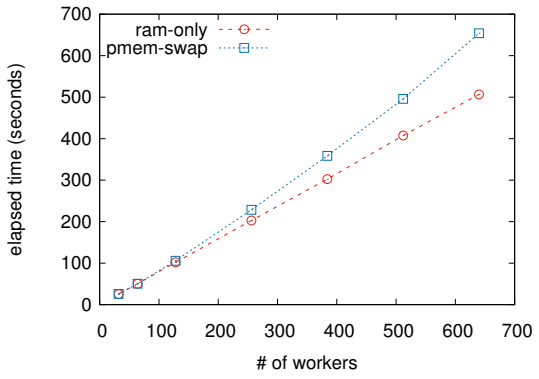


Figure 1: Elapsed time of Qsort

swap grows wider. This gap implies a software bottleneck in swapping and a foreseeable performance drop in the "small memory - fast swap" structure suggested by Y. Park et al. in [12]. We then focus on reducing swap operations by revisiting page replacement algorithm.

1.2 Our Contributions

In this paper we aim to maximize hit ratio of page replacement algorithm through analysis of reference history. In previous researches, page replacement overhead is strictly emphasized because

1. victim has to be selected in a very short period of time to prevent from becoming a performance bottleneck.
2. memory insufficiency is de facto reason behind page replacement, adding memory overhead is equivalent to creating more page faults.

However, the evolvement of multi-core processors and high-speed persistent memory mitigates constant page replacement overhead whilst latency caused by swapping is linear to size of memory allocated. We have then decided to focus on hit ratio and developed our algorithm by the assumption that maximizing hit ratio achieves better performance than minimizing overhead in terms of swapping. By analyzing a page's reference history, we are able to locate and evict the least demanded page in the page frames. Our contributions can be summarized as follows:

1. We introduce a log-based page replacement algorithm following the assumption that data pages tend to keep its frequency pattern throughout program life cycle.
2. We propose an equation that accurately estimates threshold size for logging so that victim suggested by threshold-sized log is no worse than by a full-sized log.
3. We present hotness attribute that mimics hot pages and their frequency accordingly as an input to synthesize artificial reference traces.

The rest of this paper is organized as follows. In Section 2 we reviewed relevant researches on page replacement problem. A detail description of our proposed log-based algorithm, along with hotness-featured artificial traces is explained in Section 3. In Section 4 we depict our evaluation results and verify the proposed equation of window-size estimation. Finally, a conclusion is presented in Section 5.

2. RELATED WORKS

Page replacement as an online problem, its performance is limited by an optimized but unattainable upper bound defined in Bélády's MIN [4] algorithm. To reach optimal hit ratio, a full sequence of future references are acquired and calculated backward. It is generally impossible and not implementable for most of the applications.

Researches on page replacement algorithm therefore aims to maximize hit ratio whilst incurring acceptable overhead. The classical researches on page replacement algorithm utilizes either frequency or recency of requested pages for victim selection. For example, "Not Frequently Used" (NFU) evicts the page that holds least reference counts, and "Least Recently Used" (LRU) evicts the one holds the oldest reference time.

An improved version of LRU was introduced in LRU-K [11] that maintains a buffer of referenced pages and evicts the page whose K-th most recent access holds the furthest position in the buffer. The LRU-2 outperforms LRU 8% at best for random access with Zipfian 80-20 distribution. Besides, there is no estimation of threshold log size being proposed.

Derived from NFU, AGING [8] creates a reference counter per page in bit field format that tracks every page accesses. On each clock tick, the bits shift rightward by one. If a page is referenced, the leftmost bit of its reference counter is set to 1. When a page in memory is to be replaced, the page holds lowest bit field value is evicted.

Modern researches of page replacement focuses on optimization for new generation storage devices. Motivated by the emergence of phase change memory (PCM), an example of persistent memory, PCM-based swap subsystem are proposed in [6] and [12]. Chen et al. proposed to reduce PCM write for PCM/DRAM hybrid memory, while Park et al. proposed a co-working paradigm of DRAM and PCM as a "small memory - fast swap" structure to reduce power consumption of DRAM.

3. PROPOSED ALGORITHM

Log-based algorithm keeps a list of reference counters and a queue-structured page log. Whenever a page is referenced, the page number is enqueued to the log's head and its corresponding counter is incremented by one. If the log is full, the page at log's tail is dequeued, and the counter is decremented by one accordingly.

Let $RC(P_k)$ denotes the reference count of Page k, N denotes number of distinct pages appeared in the log, and W denotes the length of log, then the reference counters fulfills following condition.

$$\sum_{k=1}^{N} RC(P_k) = W$$

The algorithm evicts the page that holds the lowest reference rate in the log when page replacement is needed. The detail algorithm is described in Algorithm 1.

Algorithm 1: Log-based Page Replacement Algorithm

Result: a victim page is selected
Input : Log, log of pages referenced
$TotalRef$, total number of logged references
Output: $victim$, page with lowest reference rate

```
page faults;
FrameList ← list of pages stored in memory;
CounterList ← list of page reference counter in Log;
minRR ← 1;
victim ← NULL;
Function RC(Page_k):
    return reference counts of Page_k;
End
Function RR(Page_k):
    return RC(Page_k)/TotalRef;
End
foreach Page_i in FrameList do
    foreach Page_j in CounterList do
        if victim is NULL then
            victim ← Page_j;
        end
        if Page_i equals Page_j then
            if RR(Page_j) < minRR then
                minRR ← RR(Page_j);
                victim ← Page_j;
            end
        end
    end
end
```

Algorithm 1 is entered when page faults and no empty frame is available. A list of page reference history Log and an integer of total reference counts $TotalRef$ are taken as input. The reference rate $RR(Page_k)$ of each page $Page_i$ stored in $FrameList$ is traversed and compared with each other. At the end of loop, variable $victim$ is assigned to the page with lowest reference rate and returned as output.

3.1 Hotness

It is generally impossible to accurately predict future page requests in an open system. However, we have assumed that pages being requested more frequently are more likely to be requested in the near future. This assumption also implies that not all pages are requested with equal probability, namely, there exists a hot/cold pattern in page requests. We then borrow the idea from Yoo's work [13] to synthesize page request traces with different hotness.

We have designed eight types of hotness, 10/90, 15/85, 20/80, ..., and 45/55. The former number defines number of hot pages among the page pool in percentage format, and the later defines aggregate reference rate of hot pages. Take 10/90 for example, it defines 10% of the distinct pages are hot, and those hot pages contribute to 90% of the requests in the synthesized trace. Namely, the hotness reference rate is evenly shared by the hot pages.

3.2 Page Reference Log

Most page replacement algorithms utilize a part of page reference history as a hint to "guess" which page is least likely to be referenced again. Needless to say, it is impractical to assume a unlimited reference history. We then introduce a parameter W_{thres} to evaluate threshold window size.

Threshold window size is the minimum number of logged pages that makes the algorithm performs as good as it does via a full-sized log. To find the threshold size that contains sufficient information for victim selection, we have developed an equation as listed in Equation 1

$$W_{thres} = \lceil \lceil \frac{hN}{1-h} \rceil \times hN \times M \rceil \quad (1)$$

where N denotes the number of reference counters, i.e., the number of distinct pages in the log, h denotes hotness, and M denotes hotness multiplier. A denotation table is shown in Table 1.

The equation to calculate hotness multiplier is shown in Equation 2, which represents how many times a hot page should be referenced whilst each of rest pages (non-hot pages) is referenced at least once.

$$M = (1 + \frac{hN}{F})^2 \quad (2)$$

Table 1: Denotation table

M	Hotness multiplier.
N	number of distinct virtual pages being requested.
F	number of physical page frames.
h	hotness, one of the eight hotness types in percentage format.
W_{thres}	threshold window size, number of page references being logged.

4. EVALUATION

We have simulated OPTIMAL, RANDOM, and three conventional page replacement algorithms, including LRU, NFU, AGING, along with our log-based solution for comparison. The page reference traces are generated per test run and sent to each algorithm to process. Hit ratio of each algorithm is reported at the end of simulation.

Firstly we keep window size and hotness unset to see if hit ratio grows linearly to log size in our proposed algorithm. The number of page requests being synthesized includes 10,240; 20,480; 40,960; 81,920; 163,840; 327,680 and 655,360. Traces are randomly generated per simulation cycle, and the hit ratio of each algorithm is averaged from 10 cycles. In Figure 2

an overall comparison of hit ratio versus number of reference is depicted.

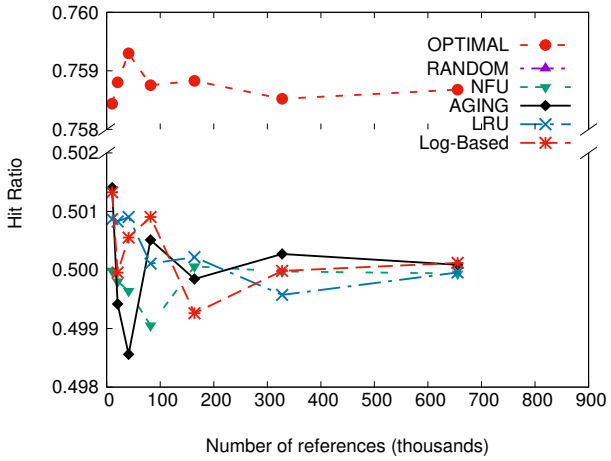


Figure 2: hit ratio versus number of references

Above all algorithms is the OPTIMAL, with 75.9% hit ratio, that defines the upper bound hit ratio of page replacement problem as explained in previous section. The rest algorithms all lie around 50%, which is about the ratio of number of page frames to number of distinct pages. In this case it's $10/20 = 50\%$. Although the difference between algorithms are so trivial that is negligible (less than 0.5%), we can still conclude from this result that hit ratio of log-based algorithm does not grow linearly to number of references.

However, in a real system, not all pages are referenced with equal rate. For example, LRU is based on the assumption that a page will be re-accessed soon after it is accessed. We thus generate traces with different level of hotness as input. In Figure 3 we compare hit ratio of algorithms with a 40/60 trace.

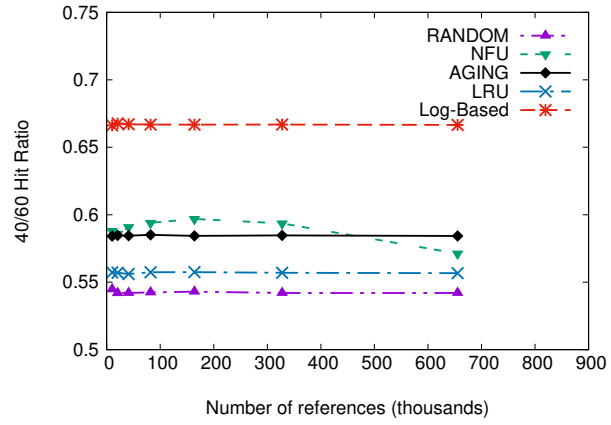


Figure 3: 40/60 hit ratio versus number of references

The simulation with 40/60 hotness shows that log-based algorithm is about 11.2% better than the most conventional LRU, and 8.3% better than AGING on hit ratio. Other than higher hit ratio, we also found out that our algorithm keeps a steady performance and draws a straight line across traces of various number of references.

We also observed that performance of all algorithms become more stable after 163,840 references. Therefore, in following simulations we fix the reference number at 163,840.

4.1 Hotness

To observe the performance change of page replacement algorithms, we defined eight types of hotness and synthesized traces accordingly. Note that we use hotness 0 to denote that no page is deliberately referenced with probability higher than others. All pages are randomly generated in the trace.

A general trend of hit ratio versus hotness shown in Figure 4 is that the hit ratio drops when hotness grows. In 10/90 hotness, a small number of hot pages being referenced for most times. This trace pattern implies that a small portion of page frames are repeatedly referenced. Of such a larger portion of page frames are left for "good victims," pages that are rarely referenced. All algorithms reaches 90% hit ratio.

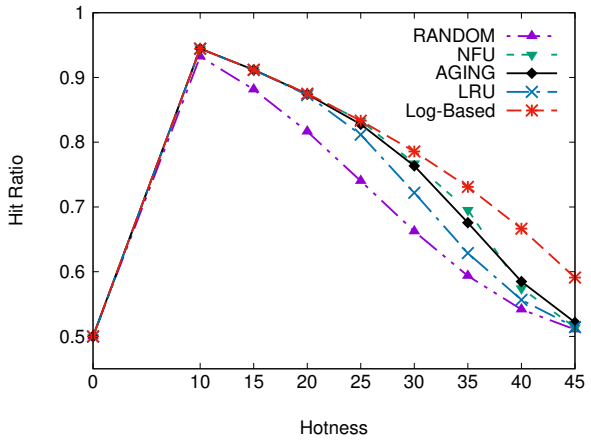


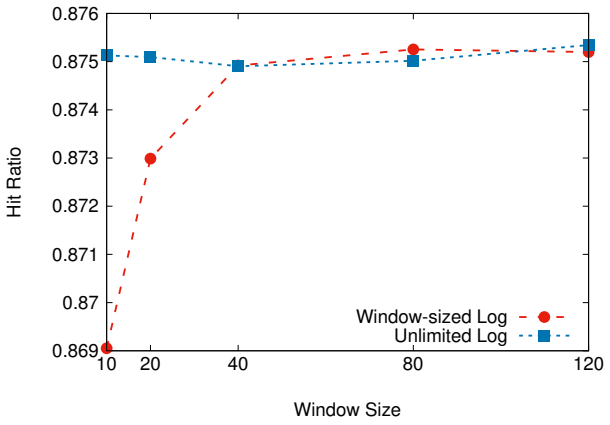
Figure 4: hit ratio versus hotness

The hit ratio drops for all algorithms when hotness grows. Because as the number of hot pages increases, the number of page frames for "good victims" decreases. We also observed that log-based algorithm starts to outperform other algorithms since hotness 30/70. It is 6.38% better than LRU, 2.23% better than AGING, 1.84% better than NFU.

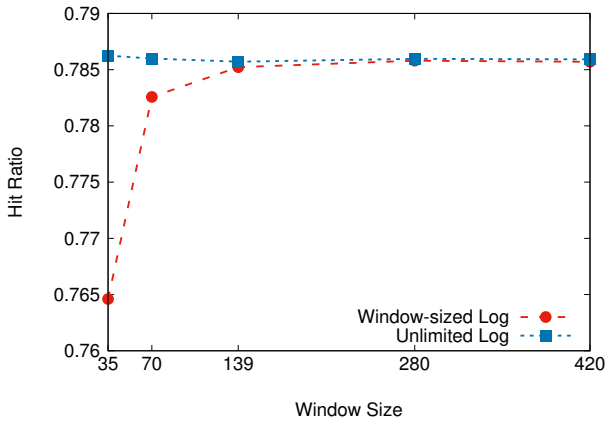
The most improvement lies in hotness 40/60, log-based algorithm outperforms AGING for 8.18%, NFU for 9.21%, and LRU for 11%. This improvement declines, however, in hotness 45/55, which is 7.65% better than LRU, 6.91% better than AGING, and 7.7% better than NFU. This is because when hotness gets closer to 50/50, where half of the pages shares half of the references, no page is hot. Therefore the hit ratio slowly drops back to about 50%, the same as hotness 0.

4.2 Threshold Size

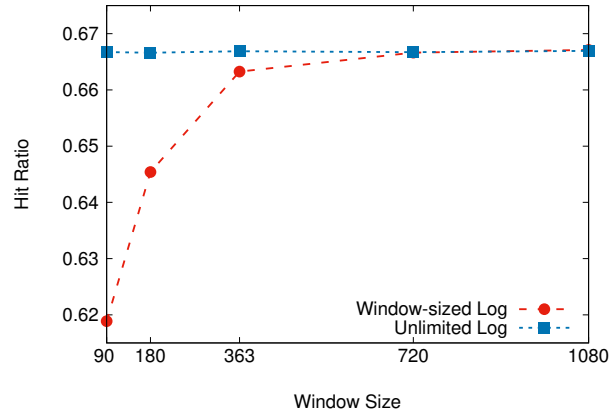
In previous simulations, we set no limit to log size and every requested page reference counts. However, the assumption



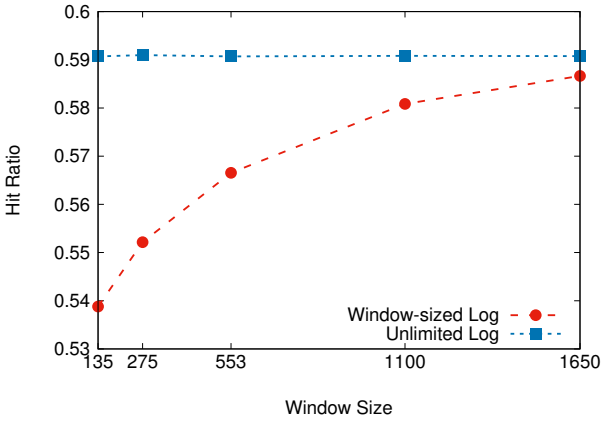
(a) 20/80



(b) 30/70

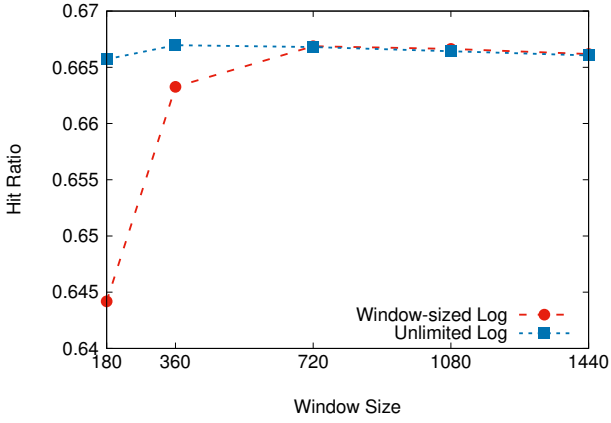


(c) 40/60

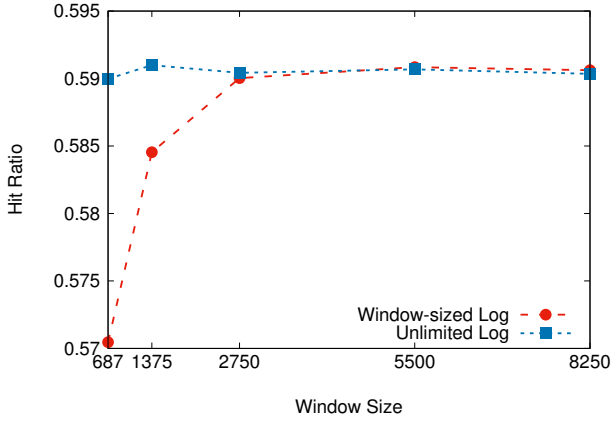


(d) 45/55

Figure 5: hit ratio versus window size on different hotness



(a) 40/60



(b) 45/55

Figure 6: hit ratio versus threshold size in warm zone

of a unlimited log or a very huge log is impractical. To find out an efficient threshold log size, we have defined a threshold table based on Equation 1. A series of simulations is designed to evaluate log-based algorithm against window sizes, along with full log as a comparison. The simulated window sizes include $W_{thres}/4$, $W_{thres}/2$, W_{thres}, $2W_{thres}$, and $3W_{thres}$.

Observed from Figure 5a and Figure 5b, we found that hit ratio reaches a plateau at the threshold we predicted. Moreover, the hit ratio remains at the same level at $2W_{thres}$ and $3W_{thres}$, which implies that our equation has successfully predicted an efficient threshold size for page reference log.

In Figure 5c, there shows a gap between hit ratio of W_{thres} and of full log. Nevertheless, hit ratio reaches the plateau at both $2W_{thres}$ and $3W_{thres}$, which implies that $2W_{thres}$ might be a better choice for log size. The threshold prediction deviates in hotness 45/55 as well in Figure 5d. The hit ratio of $3W_{thres}$ is still about 0.4% lower than full-sized log.

We reviewed the reference rate of each page in 40/60 and 45/55 traces and found that the boundary between hot and cold pages is blurred. The difference between reference rate of hot and cold pages is about 4.2% in hotness 40/60, and about 2% in hotness 45/55. This observed narrow difference makes hot pages warm. We then define hotness 40/60 and 45/55 in a "warm zone" where threshold size is approached by simulations.

For window size in the warm zone, we found $2W_{thres}$ for hotness 40/60 and $5W_{thres}$ for hotness 45/55 meets the requirement for threshold. The hit ratio difference between threshold-sized and full-sized log in hotness 40/60 is less than 0.007%, and in hotness 45/55 is less than 0.04%. The simulation result is shown in Figure 6.

5. CONCLUSION

Swap subsystem is introduced to mitigate problems of insufficient memory. In this paper we have pointed out that by the most advanced storage technologies, bottleneck of swapping has transferred from hardware to software.

A new page replacement algorithm that utilizes frequency information of logged page reference history is thus proposed in this paper. The simulation shows that our algorithm achieves better hit ratio under synthesized traces of 25/75, 30/70, 35/65, 40/60 and 45/55 hotness. It improves hit ratio by 11% than LRU, 9.21% than NFU, and 8.18% than AGING at best respectively.

We have also proposed an equation that predicts threshold size of log efficiently. The simulations have verified that the performance difference between threshold-sized log and full-sized log is indistinguishable. We also pointed out that there exists a "warm zone" in threshold prediction where difference between hot and cold pages is blurred so that our prediction is slightly deviated. However, an adjustment to threshold calculation for "warm zone" traces can be approached by multiplying threshold W_{thres}.

6. FUTURE WORKS

The potential extension of this paper includes 1) implement more advanced page replacement algorithms into our simulator for comparison, 2) integrate our log-based algorithm into a real system, and 3) refine our algorithm to reduce overhead.

7. REFERENCES

[1] pmem.io, Persistent Memory Programming. http://pmem.io/.

[2] Stress-ng. http://kernel.ubuntu.com/~cking/stress-ng/.

[3] M. D. Assunção, R. N. Calheiros, S. Bianchi, M. A. Netto, and R. Buyya. Big Data computing and clouds: Trends and future directions. *Journal of Parallel and Distributed Computing*, 79-80:3–15, 2015. Special Issue on Scalable Systems for Big Data Management and Analytics.

[4] L. A. Bélády. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[5] C. P. Chen and C.-Y. Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information Sciences*, 275:314 – 347, 2014.

[6] Chen, Kaimeng and Jin, Peiquan and Yue, Lihua. A Novel Page Replacement Algorithm for the Hybrid Memory Architecture Involving PCM and DRAM. In *Network and Parallel Computing: 11th IFIP WG 10.3 International Conference, NPC 2014, Ilan, Taiwan, September 18-20, 2014. Proceedings*, pages 108–119, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[7] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[8] M. Z. Farooqui, M. Shoaib, and M. Z. Khan. A Comprehensive Survey of Page Replacement Algorithms. *IJARCET), January*, 2014.

[9] S. Mittal and J. S. Vetter. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, May 2016.

[10] O. Mutlu. Memory scaling: A systems architecture perspective. In *2013 5th IEEE International Memory Workshop*, pages 21–25, May 2013.

[11] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *SIGMOD Rec.*, 22(2):297–306, June 1993.

[12] Y. Park and H. Bahn. Efficient Management of PCM-based Swap Systems with a Small Page Size. 15:476–484, 2015.

[13] Y.-S. Yoo, H. Lee, Y. Ryu, and H. Bahn. Page Replacement Algorithms for NAND Flash Memory Storages. In *Proceedings of the 2007 International Conference on Computational Science and Its Applications - Volume Part I*, ICCSA'07, pages 201–212, Berlin, Heidelberg, 2007. Springer-Verlag.