2016 45th International Conference on Parallel Processing Workshops

# Memory-Aware Management for Heterogeneous Main Memory using an Optimization of the Aging Paging Algorithm

Gal Oren
Department of Computer Science,
Ben-Gurion University of the Negev,
POB 653, Be'er-Sheva, Israel
Department of Physics,
Nuclear Research Center-Negev,
P.O.B. 9001, Be'er-Sheva, Israel
orenw@post.bgu.ac.il

Leonid Barenboim
Department of Mathematics and Computer Science,
The Open University of Israel,
P.O.B. 808, Ra'anana, Israel
leonidb@openu.ac.il

Lior Amar
Parallel Machines Ltd
lior@cs.huji.ac.il

***Abstract*—In the near future new technologies will make it possible to enlarge the main memory layer in the current memory hierarchy using devices with small cost and access penalties, such as the Storage Class Memory (SCM). In order to use those technologies as efficiently as possible, we need to understand how the developer and the operating system can get the best performances while managing a new heterogeneous main memory, which consists of multiple types of memory with different volumes and different access speeds. We found that the most reasonable way to introduce these new technologies into any usable memory system would be by using a new automated layer that selects the most appropriate memory levels for allocating space in the memory complex, and that moves data between memory levels of the memory complex for optimizing performance in the fashion of paging algorithms. Specifically, we discovered that this memory management is optimized using a modification of the Aging algorithm (a directive of the LRU concept) – a modification which can improve the access speed of the heterogeneous main memory by about 75%, and that manages to achieve the same or better Hit / Miss ratio in almost all cases in comparison to the current alternatives.**

***Keywords-Memory Hierarchies, Main Memory, Paging, Aging Algorithm, Storage Class Memory, Heterogeneous System.***

## I. INTRODUCTION

Memory and storage are often assumed to be unsophisticated, 'flat' resources, with simple properties, such as a constant access time. Over the years this assumption has been proven to be wrong, and understanding of the memory hierarchy could be useful in order to enhance the performance of an algorithm or a data structure. For example, the Storage Class Memory (henceforth, SCM) is a new technology which represents a new hybrid form of storage and memory with unique characteristics, meaning a memory which is non-volatile, cheap in per bit cost, has fast access times for both read and writes using cache line access, and is solid state. Also, the SCM is supposed to have different versions with different access speeds and different volumes, meaning that it might be possible to add different SCM devices to the memory hierarchy as an extension of the RAM, and manage this enlarged heterogeneous main memory using special algorithms. Our hypothesis is that achieving appropriate transferability between these new heterogeneous main memory levels may be possible using ideas of algorithms employed in current virtual memory systems, and that an efficient memory-aware adaptation of those algorithms to a heterogeneous main memory is achievable.

In order to reach the conclusion that our hypothesis is correct, we investigated various paging algorithms, and found the ones that could be adapted successfully from a standard memory hierarchy to a hierarchy with heterogeneous main memory. We discovered that using a memory-aware adaptation of the Aging paging algorithm results in the best performances in terms of Hit / Miss ratio and access speed.

In this paper we argue that memory management for future heterogeneous main memory should include our memory-aware optimization of the Aging paging algorithm. We focus on the platforms that should be part of the future heterogeneous main memory (such as SCM technology) and on the techniques and algorithms that should allow to optimize current and future heterogeneous main memory. We show that an optimization of the Aging paging algorithm makes it memory-aware with a low calculation cost. To support our argument, we present in the second part of the paper a detailed explanation about the different approaches to the management of the new heterogeneous main memory. Afterwards, in the third part of the paper, we present the novel memory-aware optimization to the Aging paging algorithm. Finally, in the fourth part of the paper, we present the benchmarks that support our hypothesis, and in the last part, we present our conclusions and the future work plans.

## II. BACKGROUND

### A. Memory Hierarchy Awareness

Understanding the memory hierarchy can be useful in order to enhance the performance of an algorithm or a data structure [1]. Algorithms and data structures that adjust to a specific memory organization are known as *memory-aware* or *memory-conscious*. Algorithms and data structures that do not take into consideration memory parameters and hierarchy are called *memory-oblivious*. Design of memory-aware algorithms requires awareness of the memory

2332-5690/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPPW.2016.29

hierarchy, as the latency and bandwidth penalty between the different levels of the memory is significant. In the following chapters we will refer to algorithms as *memory-aware algorithms* if they are taking the memory levels in account as an integral parameter that affects the total performance of the algorithm directly. We will refer to algorithms as *memory-oblivious algorithms* if they are not taking the memory levels in account as an integral parameter, and simply see the memory levels as a 'flat' resource, in which each level of memory is treated only as an extension of the previous one, without specified differentiation between them.

### B. Previous Work

Operating systems (henceforth, OS) implement the virtual memory mechanism that extends the working space for applications, mapping an external memory file (page file) to virtual addresses. This idea supports the Random Access Machine model [2] in which a program has an infinitely large main memory. With virtual memory, the application does not know where its data is located, whether in the main memory or in the page file. This abstraction does not have large running time penalties for simple sequential access patterns: The OS is even able to predict them and to load the data ahead of time.

For more complicated patterns, especially in High-Performance Computing (henceforth, HPC), these remedies are mostly not useful and even might be counterproductive. The page file is accessed very frequently, the executable code can be swapped out in favour of unnecessary data, and the page file is highly fragmented and thus many random I/O operations are needed for scanning. Therefore, in this scenario, there are two options to resolve the problem: the first option is increasing the Hit / Miss ratio and the access speed in the virtual memory mechanism, and the second option is an explicit handling of memory accesses. In this scenario, the applications and their underlying algorithms and data structures should care about the pattern and the number of memory accesses (I/Os), which they cause.

Several simple models have been introduced for designing I/O-efficient algorithms and data structures. The most realistic model is the Parallel Disk Model (PDM) of Vitter and Shriver [3]. In this model, I/Os are handled explicitly by the application. The most common implementation of the PDM model can be found at the STXXL project [4]. The core of STXXL is an implementation of the C++ standard template library STL for external memory (out-of-core) computations, i.e., STXXL implements containers and algorithms that can process huge volumes of data that only fit on disks. While the compatibility to the STL supports ease of use and compatibility with existing applications, another design priority is high performance. The performance features of STXXL include transparent support of multiple disks, variable block length, overlapping of I/O and computation, and prevention of OS file buffering overhead.

### C. Future Memory: Storage Class Memory

Storage is considered to be a mechanical HDD that supplies virtually unlimited capacity when compared to RAM, and it is also perpetual, which means that data is not lost if the computer happens to crash or disconnect from electricity. The issue with hard drives is that in various situations they are unable to supply data to applications with the sufficient speed, because of their mechanism and access fashion [1].

Storage Class Memory (SCM) [5] proposes to minimize or even close the widening gap between CPU processing speeds, the need to rapidly transfer big data blocks, and the read-write speeds suggested by HDD reliant systems. The SCM, widely known as Persistent Memory, is a technology which represents a new hybrid form of storage and memory with unique characteristics, meaning a memory which is non-volatile, cheap in a per bit cost, has fast access times for both read and writes using cache line access, and is solid state [6]. Also, the SCM is supposed to have different versions with different access speeds and different volumes, meaning that it may be possible to add different SCM devices to the memory hierarchy as an extension of the RAM, and manage this enlarged main memory using special algorithms, as the PDM model manages the disk complex using STXXL library.

The SCM has a unique mechanism. Created out of flash-based NAND, SCM is a new form of storage that can provide a middle step between high-performance RAM and cost-effective HDDs. It may very well provide read performance analogous to RAM (perhaps even better in some cases), and write performances that are significantly faster than HDD technology (factors of hundreds better than HDD and even beyond). Also, it is predicted that the production costs of SCM and HDD will be broadly similar by the end of this decade [7].

These new SCM devices connect to memory slots in a server and are mapped and accessed in the same fashion as the memory, even though they are slightly slower, and they can be addressed atomically at either the byte or the block level, unlike previous eras of storage technology. The SCM can be used directly as execution memory or data storage memory. The current SCM products include the improved Flash [8], the Phase Change Memory (PCM) [9], the Magnetic RAM (MRAM) [10], the Solid Electrolyte RAM – Nano-Ionic RAM [11], the Ferroelectric RAM (FRAM) [12] and the Memristor [13].

OS are likely to use the SCM as either very fast block storage devices formatted by file systems and databases, or as direct memory mapped "files" for next generation of programs. In the near future the SCM is predicted to modify the form of programs, the access form to storage, and the way that storage devices themselves are built. Therefore, a combination between SCM technology and the existing RAM, using a new memory allocation manager (henceforth, MAM) that will act like STXXL – but using cache line access fashion – will be likely to achieve a new level of performance for memory-aware data structures.

We employ the following model. The main memory with different kinds of memory speeds and volumes is modelled by a collection (sorted according to memory speeds) of N arrays, such that each array represents a memory level, where level 1 is the fastest and level N is the slowest. Each level is assigned a set of properties to specify its speed and size.

### D. Data Structures Problem in Usage of the Memory Allocation Manager

There are clear benefits of using a MAM at the design and implementation stage of any data structure which needs to handle massive data sets. For example, given that the keys are arranged in a list form, there is an option that the first immediate key will be stored in the fastest memory available, and as far as the list expands, the other keys will be stored in slower memories.

Therefore, it seems reasonable to re-modify the data structures to use those kinds of paradigms using a MAM, as in Fig. 1. However, this kind of solution arises a problem, which is an inherent part of this solution itself. The reason for this problem is that a MAM require not only to re-modify the memory platform and the access to it, but obviously also to re-write the memory-oblivious codes that is currently based on transparent memory access. The chances that such a significant re-write would be implemented in current codes using HPC platforms – the primary target group which need this massive enlargement of the main memory – is quite low. Most of the centres which use large computer clusters have programs that are intended to be operational for ages and are very big, complex and sustainable. Hence, the solution of rewriting the whole code and redefining the data structures to use a MAM is not possible nor plausible, and can be efficient only for several specific applications that are written today, yet not for past applications.

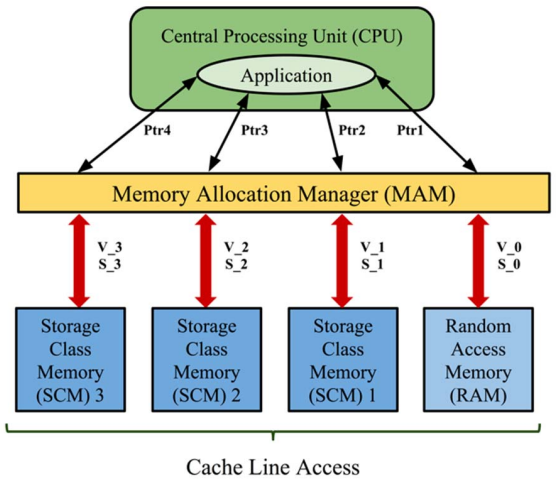


Figure 1. An Example of a Memory Allocation Manager (MAM) Diagram of Usage with Different Types of Memory.

Therefore, an implementation of a heterogeneous main memory management should not incur any changes in the data structures. This premise means that the most reasonable way to introduce the SCM into any usable memory system would be by implementing an automated and not code explicit version of a MAM using an artificial intelligence technique which can understand how to master the data in a memory-aware fashion. Thus, introducing MAM concepts and the multiple main memory levels awareness into the classic paging algorithms can be a good solution, which will not be a big performance compromise, but a modest one. This idea is achieved by introducing a new automated layer that selects the most appropriate memory levels for allocating space in the memory complex, and that moves data between memory levels of the memory complex for optimizing performance in the fashion of paging algorithms.

## III. Page Replacement Algorithms Implementation for Heterogeneous Main Memory

### A. Current Memory Management Concept and Mechanism

Computers often have five memory levels with different properties [1]. Three of these levels dwell on the processor chip, one level is the RAM memory and one level is the storage memory, such as SSD or HDD. The levels on the processor chip are referred to as L1 cache, L2 cache and L3 cache. L1 cache is a rather small piece of memory with extremely high access time, used directly by the processor. L2 cache is slightly slower and vastly larger than L1 cache. L3 cache is slower than L2 and L1 caches, and it is shared by all cores. Also, it is worth mentioning that there are architectures with an additional cache memory level, L4, which may result better performance than sole enlargement of the L3 cache level [14].

However, because the amount of data that maps to a cache section is generally much larger than the associativity of the cache, a designated replacement policy necessarily needs to determine which data to evict when a cache miss occurs [1]. Unfortunately, sufficiently precise documentation of the specific logical organization of the memory hierarchy is seldom available publicly, and the current knowledge on the different cache management policies is base on specific reverse engineering simulations [15]. Nonetheless, the current page replacement algorithms concepts are well known [16]. Replacement policies try to identify which data can be a proper candidate for eviction, by basing their resolutions on the chronology of the memory accesses. Distinguish replacement policies of this kind are Least Recently Used (LRU) and Least Frequently Used (LFU); pseudo-LRU (PLRU), an efficient variant of LRU (like the Aging policy); First-In First-Out (FIFO), also known as Round Robin; and Not-Recently Used (NRU) [16].

Either way, no matter which paging algorithm is used, it is known that an eviction from L1 forwards the cache line into L2; This obviously means a new space has to be made in L2; This in turn probably will push the data into L3 and eventually into the main memory [16]. This mechanism is also known as the model for an *exclusive cache* [17]. Other mechanisms realize *inclusive caches* where each cache line in L1 is also present in L2, and therefore an eviction from

L1 is significantly faster [17]. In SMP and NUMA architectures cache balancing algorithms manage to create a constant state in which the pressure on the caches is equal [18], and there is also a usage of paging algorithms and replication and migration algorithms [19]. Either way, the CPUs are permitted to manage the caches in any form as long as the memory model specified for the processor architecture is not modified [16].

### *B. Usage of Paging Concept in a Heterogeneous Main Memory*

Our hypothesis is that achieving a transferability between memory levels may be possible using ideas of algorithms employed in current virtual memory system, and that the adaptation of those algorithms from a standard memory hierarchy to a heterogeneous main memory may be possible. In that notion, each virtual memory page is a data entry (or entries), and each of the virtual memory swaps is done as a MAM would do if it would have supplied with knowledge of the appropriate levels. However, our approach is different from that of the paging algorithms used for cache hierarchy, specifically, cache algorithms simply transfer data to an immediate lower memory level when the current memory level is full, till the data reach to the main memory or evicts to the storage.

In contrast, our management algorithms for heterogeneous main memory will allow data to flow from one level to another freely, and not only the way down in the hierarchy or directly to the most upper level when referenced. Also, because the heterogeneous main memory handles data that is not urgently needed as data which remains in the cache, there is no need to use the same algorithmic simplicity of the cache mechanism, which simply evicts pages to the next lower level when the current level reaches its capacity. Instead, it is plausible to evict the data to some specific memory level based on extra knowledge that the OS already has. This approach, however, is not in use in the cache management algorithms because of the time overhead that cannot be tolerated in the upper levels of the memory which reside near the CPU [16].

In order to reach a conclusion that our hypothesis is correct, we clarified which of the paging algorithms can be adapted successfully from a standard memory hierarchy to a heterogeneous main memory using the ideas above, and after thoroughly investigating the current paging mechanism and the main paging algorithms [20], we found the LRU-NFU algorithms – and specifically their Aging derivative – is the best match to our goals.

### *C. The Memory-Aware Aging Page Replacement Algorithm for Heterogeneous Main Memory*

The Aging algorithm is a modification of NFU algorithm which makes it possible to simulate LRU algorithm quite well. Instead of only incrementing the counters of pages referenced, the variation has two parts: First, the counters are shifted right once before the R bit is inserted, i.e., there is actually a division by 2 of the represented decimal number. Second, the R bit is inserted to the leftmost bit, instead of inserting it to the rightmost bit. For instance, if a page has referenced bits 1,1,0,0,0 in the past 5 clock ticks, its referenced counter looks as follows: 10000000, 11000000, 01100000, 00110000, 00011000. When a page fault occurs, the page whose counter is the lowest is removed. It is clear that a page that has not been referenced for about K clock ticks will have K leading zeroes in its counter (like the referenced counter in the example at the fifth clock tick which has 3 leading zeroes after 3 non-referenced clock ticks), and therefore will have a lower value than a counter that has not been referenced for K-1 clock ticks.

A transition of the memory-oblivious Aging algorithm to be a memory-aware algorithm, for usage in a heterogeneous main memory model, can even add another level of sophistication, especially because of the existence of a linear proportion between the degradation of the referenced bits and the amount of time that a specific page has not been in use. In this hypothesis there is an interesting phenomenon; specifically, there is a possibility to create a **direct link** between the amount of zeroes in the beginning of the page referenced bits to the level of memory that that page should be evicted to according to its usage proportion (L in Formula (1)). This is achieved using a calculation which should take only few floating-point operations, and which is based on information that the OS already holds. Based on the knowledge that the amount of zeroes points to the amount of unreferenced past clock ticks – and therefore on the page aging status – it would be wise to evict the page straight to its proportionate level of memory base on the following Formula (1):

$$L = \left\lceil \frac{\text{Amount of Initial Zero Bits}}{\left\lceil \frac{\text{Amount of Reference Bits}}{ML} \right\rceil} \right\rceil \qquad (1)$$

For instance, if a page has referenced counter which equals to 00001000, while there are 3 levels in the memory complex (ML), the proportionate level of memory (L) that this page will be transferred to at the update stage of the OS pages is the second level.

Therefore, a life cycle of a page should be as in the following route: First, the page is inserted into the memory hierarchy (using *Insertion* function below; Algorithm 1); then, depending on its aging status, it is 'diffusing' to lower memory levels in the complex hierarchy (using *Update* function below; Algorithm 2). It is worth noting that if the page is being referenced, it is redirected right to the first level (also using the *Update* function). Hence, by forming a dynamic pyramid hierarchy of both page and memory necessity it becomes possible to get significantly better performances for the Aging paging algorithm in a heterogeneous main memory.

**Algorithm 1**

- *Set* memory levels to N.
  /* ML = N */
- *Set* current memory level pointer to the highest.
  /* L = 1 */

1. **Insertion** of a new page P:
    1.1. *If* the current memory level pointer is outside the lowest memory level /* L > ML */:
        1.1.1. *Return* False. /* Recursion Termination */
    1.2. Call to the page P.
    1.3. *If* the page P exists in the memory / storage:
        1.3.1. *Check* if placing in the L-level of memory is possible.
        1.3.2. *If* placement possible:
            1.3.2.1. *Place* page P at the L-level of memory.
            1.3.2.2. *Return* True. /* Recursion Termination */
        1.3.3. *Else If* placement impossible:
            1.3.3.1. *Find* the page with the lowest referenced counter which has not presently been referenced:
                1.3.3.1.1. tmpP = *Remove* and *Fetch* the page with the lowest referenced counter.
                1.3.3.1.2. *Place* the page P instead of the removed page.
                1.3.3.1.3. *Calculate* L by Formula (1).
                1.3.3.1.4. *Do* **Insertion** of the removed page tmpP to level L. /* Recursion Invocation */
            1.3.3.2. *If* no such page has been found:
                1.3.3.2.1. *Do* **Insertion** of page P to level L+1.
    1.4. *Else If* page does not exist in the memory / storage:
        1.4.1. *Return* False. /* Recursion Termination */

**Algorithm 2**

1. **Update** of an existing page (*by the OS*):
    1.1. *If* Read / Write action performed on the page:
        1.1.1. *Set* R bit to 1 (R = 1).
    1.2. *If* clock interrupt:
        1.2.1. *Right Shift* one bit to the page counter.
        1.2.2. *Add* the R bit to the leftmost bit of the page counter.
        1.2.3. newL = Calculate L by Formula (1).
        1.2.4. *If* newL ≠ L:
            1.2.4.1. tmpP = *Remove* and *Fetch* the page with the lowest referenced counter.
            1.2.4.2. *Do* **Insertion** of the removed page tmpP to level newL.

It is worth noting that an implementation of the suggested algorithm with an update at every clock tick can cause a waste of precious CPU cycles. Also, current page replacement paradigms are based on a linear time complexity. Due to such practical concerns, Linux OS for example, implements Second Chance replacement algorithm with 2-Q [21] which neither requires repetitive calculation of page's ages at every clock tick, nor has a high-complexity page replacement cost. Therefore, in order to enhance performances on the one hand without degrading them on the other hand, it is possible to use the algorithm as a secondary assisting memory management algorithm. Also, in order to minimize the overhead of this algorithm we suggest an update only every several cycles, which will be in proportion to the size of the working set of the process.

### *D. Simulative Implementation of Memory-Aware Paging Algorithms*

In order to verify our hypothesis regarding the generalization needed to transfer the memory-oblivious Aging page replacement algorithm to be applicable to multiple levels of memory in a memory-aware fashion, we created a simulator that is able to run on any computer, and that is able to simulate a situation in which frames of memory are managed and mapped to specific levels of memory in a memory-aware or in a memory-oblivious fashion, based on the chosen algorithm. The simulator works in a uniform memory access (UMA) and its inputs can be selected either randomly or explicitly. In our benchmarks we used a strict model of random only inputs to verify our hypothesis.

### *E. Algorithms Benchmark*

In order to verify our hypothesis regarding the beneficence of using the modified memory-aware Aging page replacement algorithm in heterogeneous main memory, and especially when this memory complex is a complex of standard RAM and different types of Storage Class Memory (SCM), we need to verify two main hypotheses:

- First, that the memory-aware algorithm is resulting in an equally efficient Hit / Miss ratio as the memory-oblivious Aging algorithm when it implemented on heterogeneous main memory using the explicit cache mechanism. The explicit cache mechanism is the point of reference to the new algorithm because unlike the implicit cache mechanism it is not duplicating data to a lower level, and clearly keeps the different levels in the hierarchy independent.
- Second, that the memory-aware algorithm is resulting in a significantly better access speed than

the memory-oblivious Aging algorithm when it is implemented on heterogeneous main memory using the explicit cache mechanism. This parameter should show that the optimization to the classic memory-oblivious Aging algorithm actually manages to transfer the different pages to their designated memory levels in the memory complex based on the proportionate frequency of their usage, and that this redirection of pages actually manages to move the more needed pages to a better access-time levels in the memory complex, and by that to achieve better total performances.

Therefore, we tested and compared the two types of algorithms – the memory-aware Aging and the memory-oblivious Aging – on two different platforms:

1. A uniformed model of 3-level main memory which consist of a classic one-level memory (RAM only) and two extra memory levels with the same volume – for accurate comparability measurements – as the first level. Those two extra levels were simulating two different types of SCM devices: One which was 2 times slower than the RAM, and the other which was 3 times slower than the RAM.
2. A more realistic model of 3-level main memory which consist of a classic one-level memory (RAM only) and two extra memory levels with different volumes in an ascending volume hierarchy. Those two extra levels were simulating two different types of SCM devices: one which was 2 times slower but 10 times larger than the RAM, and the other which was 3 times slower but 100 times larger than the RAM.

Due to the fact that the results of the benchmarks on the two platforms were almost the same with a slight advantage to platform 2, we will present only the last results without underestimating the importance of the first ones.

### *F. Results and Analysis*

As previously mentioned, the benchmark of the algorithms, using our simulation, has been performed on two different architectures, using several parameters. The following graphs show this benchmark result, the Hit / Miss ratio and the average memory level access, which will be presented as a function of the amount of page references (R) when the amount of frames in memory (F) and the amount of unique page indexes (I) are fixed. We ranged the amount of page references in two scales: The first scale ranged from 10 till 100, and the second scale ranged from 1000 till 1 million. The purpose of those two scales is to examine the performance of the simulation in normal usage scale and in intense HPC scale, respectively.

In order to verify our hypothesis, we compared the new optimized memory-aware Aging algorithm with a memory-oblivious Aging algorithm which is not aware of the multiple levels of the memory complex as we suggested in this paper. The most reasonable way to implement this algorithm was by using the same technique the exclusive cache mechanism manages the transferability of data between the different cache levels, i.e., by transferring data from one level to a lower one when the current level is full, and not to a specific level based on prior knowledge, as the mechanism of the optimized memory-aware Aging algorithm does.

Therefore, we re-examined the Hit / Miss ratio as a function of the amount of page references (R) using platform 2 for both algorithms and discovered that the Hit / Miss ratio as a function of the amount of page references (Fig. 2) was finally almost the same, meaning that despite the fact that the optimized memory-aware Aging algorithm is transferring data to other levels even before the memory level is full, there is no negative impact on the Hit / Miss ratio.

Furthermore, and most importantly, we examined the access speed to the memory complex – in this case by the average memory-level access – as a function of the amount of page references (R) (Fig. 3), and discovered that the memory-aware Aging algorithm is yielding about 75% improvement in the access speed over the memory-oblivious Aging algorithm, as evident from the lower average access levels in comparison to the memory-oblivious algorithm (Recall that the memory levels are ordered according to their speed, and lower levels are faster). This means that although the Hit / Miss ratio in both algorithms is almost the same in most cases, there is a clear advantage to the memory-aware Aging algorithm, as it resulting in much better performances than the memory-oblivious Aging algorithm using heterogeneous main memory.

In addition to the previous results, we examined whether the optimization mechanism that we previously suggested and tested yields the same performances when pages are redirected to a different level in the memory complex than the level that Formula (1) suggests. Our hypothesis is that Formula (1) is the most appropriate formula, and using a different one would reduce performance. In order to verify this assumption, we examined the results of the memory-aware Aging paging algorithm using platform 2 with intentional modification where there is **no direct link** between the amount of zeroes in the beginning of the page referenced bits to the level of memory that that page should be evicted to.

In order to test the hypothesis that by forming a dynamic pyramid hierarchy of both page and memory necessity it was possible to get the best performances for a paging algorithm in a multi-level main memory, we modify the behaviour of the algorithm to select different levels rather than the correct direct levels. Specifically, if a page at level 1 was directed towards level 2 in our original algorithm, it is actually redirected to level 3 and vice versa. Afterwards, we re-examined the Hit / Miss ratio of the modified algorithm as a function of the amount of page references (R) (Fig. 4) and discovered, unsurprisingly, that there was a loss in performance in comparison to our original algorithm. This loss can be explained by eviction of pages from the heterogeneous memory-complex although those pages were

actually more needed than the pages that ultimately remained in the memory.

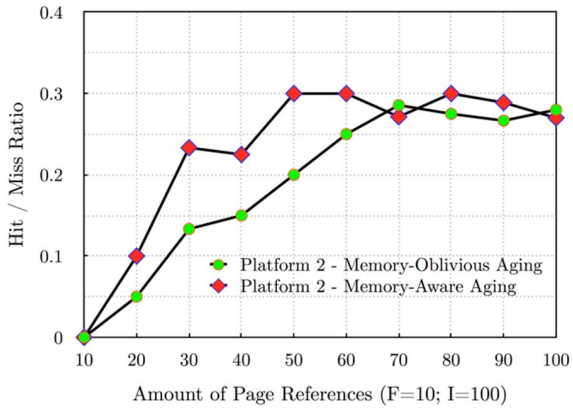


Figure 2. Memory-Oblivious vs. Memory-Aware Hit / Miss ratio as function of the Amount of Page References in a Heterogeneous Main Memory (Platform 2).

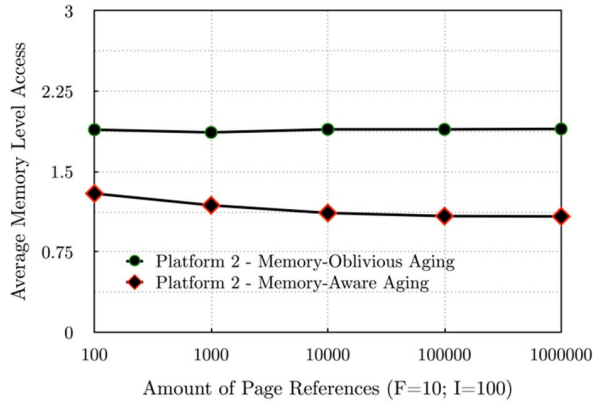


Figure 3. Memory-Oblivious vs. Memory-Aware Average Memory-Level Access as a function of the Amount of Page References in a Heterogeneous Main Memory (Platform 2).

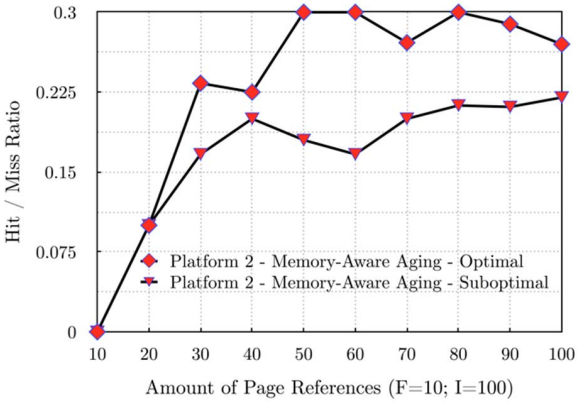


Figure 4. Optimal Memory-Aware vs. Suboptimal Memory-Aware Hit / Miss ratio as function of the Amount of Page References in a Heterogeneous Main Memory (Platform 2).

## IV. Conclusions and Future Work

Those benchmarks, results and analysis lead to the conclusion that it would be beneficial to use the memory-aware Aging paging algorithm in a heterogeneous main memory which includes SCM devices in standard computing systems as well as in HPC clusters.

This paper opens a number of prospective directions for future research. One immediate direction is to explore how the memory-aware Aging paging algorithm is reacting when the memory levels are not from the same class, and what exactly does that mean in aspects of cost, volume, memory-access fashion and access of speed. Another direction is to understand how to optimize other memory management algorithms which are not paging algorithms for beneficial usage of the heterogeneous main memory.

Finally, we also expect that in the near future the SCM invention will be a real and widespread technology, meaning that investigating actual SCM devices, applying our algorithm in managing them as part of a heterogeneous main memory, and comparing the results to the presented simulations would be a fertile ground for further research and development.

## Acknowledgments

We thank the reviewers for their useful comments.

This work was supported by the Lynn and William Frankel Center for Computer Science, and by Israel Science Foundation grant 724/15.

This research was also supported by the Open University of Israel Research Fund. Part of this work has been performed while the first-named author was a M.Sc. student at the Open University of Israel.

## References

[1] Andrew S. Tanenbaum, **Modern Operating Systems**, Prentice Hall, 4nd edition, 2014.

[2] Von Neumann, John, **First Draft of a Report on the EDVAC**, IEEE Annals of the History of Computing 4, 1993, 27-75.

[3] Vitter, Jeffrey Scott, Elizabeth A. M. Shriver, **Algorithms for Parallel Memory, II: Hierarchical Multilevel Memories**, Algorithmica 12.2-3, 1994, 148-169.

[4] Dementiev, Roman, Lutz Kettner, Peter Sanders, **STXXL: Standard Template Library for XXL Data Sets**, Softw., Pract. Exper. 38.6, 2008, 589-637.

[5] Freitas, Richard, Winfried Wilcke, Bülent Kurdi, G. W. Burr, **Storage Class Memory, Technology and Use**, In Tutorial, 6th USENIX Conference on File and Storage Technologies, 2008.

[6] Huang, C.Y., **Storage Class Memory**, Final Report – IEE5009: Memory Systems, Institute of Electronics, National Chiao-Tung University, Fall 2012.

[7] Hession David, Nigel Mc Kelvey, Kevin Curran, **Storage Class Memory**, International Journal of E-Business Development IJED 4.1, 2013.

[8] Burr, G.W., Virwani, K., Shenoy, R.S., Padilla, A., BrightSky, M., Joseph, E.A., Lofaro, M., Kellock, A.J., King, R.S., Nguyen, K. and Bowers, A, **Large-scale (512kbit) integration of multilayer-ready access-devices based on mixed-ionic-electronic-conduction (MIEC) at 100% yield**, In VLSI Technology (VLSIT), Symposium on (pp. 41-42), IEEE, 2012.

[9] Numonyx, **The basics of phase change memory (PCM) technology**, White Paper, 2010.

[10] Shenoy, R.S., Gopalakrishnan, K., Jackson, B., Virwani, K., Burr, G.W., Rettner, C.T., Padilla, A., Bethune, D.S., Shelby, R.M., Kellock, A.J. and Breitwisch, M., **Endurance and scaling trends of novel access-devices for multi-layer crosspoint-memory based on mixed-ionic-electronic-conduction (MIEC) materials**, In VLSI Technology (VLSIT), 2011 Symposium on (pp. 94-95), IEEE, June 2011.

[11] Byrne, S, **University develops PMC memory, a potential Flash killer**, myce, 1 November 2007.

[12] Freitas, Richard F., Winfried W. Wilcke., **Storage-class memory: The next storage system technology**, IBM Journal of Research and Development 52.4.5: 439-447, 2008.

[13] Chen, Y.C., Rettner, C.T., Raoux, S., Burr, G.W., Chen, S.H., Shelby, R.M., Salinga, M., Risk, W.P., Happ, T.D., McClelland, G.M. and Breitwisch, M., **Ultra-thin phase-change bridge memory device using GeSb**, In International Electron Devices Meeting (pp. 777-780), December 2006.

[14] Bezerra, Carlos Eduardo B., Cláudio FR Geyer, **A short study of the addition of an L4 cache memory with interleaved cache hierarchy to multicore architectures**, Institudo de Informatica, Universidade Federal do Rio Grande do Sul.

[15] Abel, Andrew, Jan Reineke, **Reverse engineering of cache replacement policies in Intel microprocessors and their evaluation**, Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on. IEEE, 2014.

[16] Jacob, Bruce, Spencer Ng, David Wang, **Memory systems: cache, RAM, disk**, Morgan Kaufmann, 2010.

[17] Zheng, Ying, Brian T. Davis, Matthew Jordan, **Performance evaluation of exclusive cache hierarchies,** In Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS, pp. 89-96. IEEE, 2004.

[18] Majo, Zoltan, Thomas R. Gross, **Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead**, In ACM SIGPLAN Notices, vol. 46, no. 11, pp. 11-20. ACM, 2011.

[19] Black, David L., Anoop Gupta, Wolf-Dietrich Weber, **Competitive Management of Distributed Shared Memory**, Distributed Shared Memory: Concepts and Systems 21 (1998): 73.

[20] Oren, Gal, **Optimizations of Management Algorithms for Multi-Level Memory Hierarchy**. Diss. The Open University, 2015.

[21] Johnson, Theodore, and Dennis Shasha. **2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm**. (1994).