Storage Class Memory Aware Data Management

Bishwaranjan Bhattacharjee
IBM T. J. Watson Research Center
bhatta@us.ibm.com

Mustafa Canim*
The University of Texas at Dallas
canim@utdallas.edu

Christian A. Lang
IBM T. J. Watson Research Center
langc@us.ibm.com

George A. Mihaila
IBM T. J. Watson Research Center
mihaila@us.ibm.com

Kenneth A. Ross
IBM T. J. Watson and Columbia University
kar@cs.columbia.edu

Abstract

Storage Class Memory (SCM) is here to stay. It has characteristics that place it in a class apart both from main memory and hard disk drives. Software and systems, architectures and algorithms need to be revisited to extract the maximum benefit from SCM. In this paper, we describe work that is being done in the area of Storage Class Memory aware Data Management at IBM. We specifically cover the challenges in placement of objects in storage and memory systems which have NAND flash (one kind of SCM) in a hierarchy or in the same level with other storage devices. We also focus on the challenges of adapting data structures which are inherently main memory based to work out of a memory hierarchy consisting of DRAM and flash. We describe how these could be addressed for a popular main memory data structure, namely the Bloom filter.

1 Introduction

Database systems make extensive use of main memory. There is a whole range of products where the data is main memory resident only. These include the IBM SolidDB [15], Oracle TimesTen [18], etc. Even in systems where the data is disk based, like DB2 LUW [7] and Oracle 11g [12], main memory is used for various operations to achieve acceptable performance. These operations include caching, locking, sorting, hashing, aggregations and bitmap operations. Due to the high number of transactions supported, as well as the type of operations being performed, main memory is always at a premium on a database system. Database systems have implemented a variety of ways to balance the demand on main memory, including putting in features like Self Tuning Main Memory [16, 6]. These help in getting the best usage for the limited main memory available.

The advent of Storage Class Memory [8], like flash and Phase Change Memory (PCM), opens up the possibility that with smart algorithmic changes [13], one could think of extending data structures and operations that are inherently main memory limited to work out of a memory hierarchy consisting of DRAM supplemented with

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*The majority of this work was completed while the author was an intern at IBM T. J. Watson Research Center

SCM. SCM has good random access capability that existing storage layers in the hierarchy below DRAM lack. Current SCM like flash, provide this capability in block based fashion and if existing main memory algorithms can be modified to take advantage of this, then it is possible to extend them to work beyond main memory.

In current disk based database systems a major bottleneck is the IO capability of the disks. Transaction processing systems tend to stress the IO subsystem with their short running point queries and updates. These result in random IO, which is a major drawback of hard disks. With their moving arms, a hard disk is very limited in the number of IO operations per second (IOPS) that it can produce. Even in the case of Data Warehousing, with customers running multiple streams of queries, ingests and deletes, the IO capability of the storage system tends to be a major pain point.

In these situations, SCM based or supplemented storage solutions could help. Examples include the SSD based Teradata Extreme Performance Appliance [17] or the Oracle Exadata [11] or IBM Smart Analytics Systems [14]. Given that customers may not be able to afford to place all their data on SCM based storage due to its cost, a key challenge that needs to be addressed is what data or objects should be placed in this costlier medium so that customers can get the best value for their money. In the case of database systems, the data or objects in question could be whole objects like relational tables, indexes, materialized views. Alternatively, one could selectively store pieces (e.g., pages) of these objects.

In this paper, we describe work that is being done in the area of Storage Class Memory aware Data Management at IBM. We specifically cover the challenges in placement of objects in storage and memory systems that have flash or SCM in a hierarchy or in the same level with other storage devices. We also focus on the challenges of adapting data structures that are inherently main memory based to work out of a memory hierarchy consisting of DRAM and flash. We describe how these could be addressed for a popular main memory data structure, namely the Bloom filter. We also briefly look into how these challenges and solutions will have to adapt to future SCM technologies like Phase Change Memory.

The rest of the paper is organized as follows: in Section 2 we present both a static and a dynamic approach for selective data placement in a heterogenous storage environemnt containing SCM in addition to hard disk drives; in Section 3 we describe how Bloom filters can take advantage of SCM; we conclude in Section 4.

2 Data Placement for Heterogeneous Storage

Given a collection of storage devices with vastly different performance and cost characteristics, such as hard drives and SCM, one important issue is deciding on the physical placement of the data, to achieve good performance at a reasonable cost. At one extreme, placing all data on the faster device (SCM) will enable the best performance, at a high cost. At the other extreme, keeping all data on hard drives will be the least expensive, but it may exhibit poor performance, especially if the workload causes substantial random access to the disk. So the question is whether one can obtain significant improvements in performance at a reasonable cost by placing only a fraction of the database on the SCM. To achieve such improvements, there must be some locality of access that can be exploited. Fortunately, except for extreme workloads that scan all data, such locality does exist in practice.

One may consider either a static or a dynamic approach to the data placement problem. One can either make data placement choices as part of the physical database design stage, or dynamically re-locate data as needed in the production stage. In the remainder of this section we will describe our experience with both approaches and discuss their relative benefits and drawbacks.

2.1 Static Object Placement

In order to make informed decisions about data placement, one needs to know something about the workload that will be executed on the database. However, with the typical database having hundreds of objects (tables, indexes,

materialized views, etc.), deciding on the optimal location of each object can quickly become overwhelming even for an experienced database administrator. In order to assist the DBA in this task, we implemented an Object Placement Advisor (OPA) tool [4], that generates placement recommendations based on profiling information obtained after running a calibration workload on a test instance of the database.

Specifically, the profiling information used includes, for each database object O_i, the total time spent reading and writing disk pages belonging to that object, distinguishing among random and sequential access. Using these statistics, the OPA tool estimates the total potential benefit B_i (access time savings) were this object to be placed on to SCM instead. The potential benefit is calculated given the ratio between the throughput of the SCM device relative to the hard drive for each type of access (random vs. sequential, and read vs. write). Assuming the size S_i of each object is known, or can be estimated (for the production database), the object placement problem becomes an instance of the well-known Knapsack problem, for which good heuristics exist.

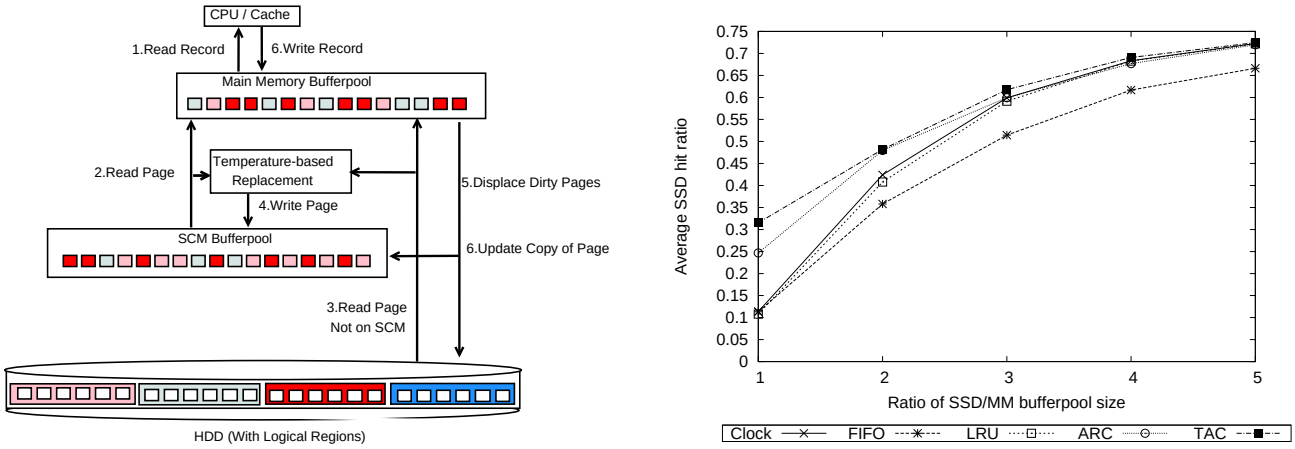
In our implementation, we used a Greedy heuristic, which delivers very competitive solutions in practice, and runs in linear time in the number of objects. Specifically, the Greedy heuristic iteratively places objects on the SCM in descending order of their average *access density* (access time savings per unit of storage) until there is no space left for the next object. To validate this approach, we implemented the recommendations computed by the OPA on the TPC-C benchmark and we compared the throughput (tpm-C) with the base database (all data on hard drive). The database layout suggested by OPA managed to deliver 10x more transactions per minute with 36% of the data placed on a consumer-grade Samsung SSD delivering 5000 IOPS (compared to about 120 random IOPS for the hard drive). For comparison, we also implemented a naive placement strategy (all indexes on SCM, all tables on magnetic disk), which achieved only a 2x throughput improvement with 23% of the data stored on SCM.

The static placement solution works best when the workload is relatively stable over time, but if the workload shifts significantly the DBA needs to re-run the OPA tool periodically and potentially move objects across storage devices to implement changes in the placement recommendations. Another limitation of this scheme is that it doesn't allow *partial* placement of objects on different devices. This means that large objects that do not fit in their entirety on the SCM will never be considered. Both of these limitations are lifted by the dynamic scheme described next.

2.2 Dynamic Temperature-Aware Bufferpool Extension

In order to enable a more flexible and adaptive placement scheme we conceptually move SCM up in the storage hierarchy, to an intermediate level between the magnetic disk(s) and main memory, effectively using it to house an extension of the main memory bufferpool [5]. Thus, the SCM-resident bufferpool extension acts like a *second-level* page cache, while the main memory bufferpool is still the primary cache. Our system architecture is summarized in Figure 1(a).

Clearly, such an arrangement has the potential to overcome both the workload adaptability and the partial placement limitations exhibited by the static scheme. However, since the access patterns for a second-level cache are quite distinct from those of a primary cache, traditional replacement policies are not best suited for governing page admission and eviction decisions. Instead, we base these decisions on the "temperature" of pages, a notion inspired from the access density used in the static scheme. Conceptually, the current temperature of a given disk page is the total time spent reading that page from disk. Since maintaining the precise access time of each database page is expensive, we group a small number of adjacent pages (currently between 8 and 64) into *regions* and maintain only the average temperature of each region. Also, instead of actually measuring the time spent, we estimate it based on the type of access (random vs. sequential) and the disk performance rating. Thus, for any given page we approximate its current temperature by the average temperature of its region. Whenever a main memory bufferpool page miss occurs, we consult the SCM bufferpool extension, and, if the page is currently cached there, we serve it (item 2); otherwise, we read it from the hard drive (item 3), and based on its region temperature, decide whether we should cache it in the SCM bufferpool. A page p is only cached in the SCM



(a) System Architecture

(b) SCM bufferpool hit ratio comparison

Figure 1: Dynamic Bufferpool Extension.

bufferpool if its temperature exceeds that of the coldest page q currently cached there, in which case q will be evicted and p stored in its place (item 4). This way, for any given workload, the SCM bufferpool content will automatically converge to the hottest set of K pages, where K is the SCM capacity in pages. To account for changes in the workload, the temperatures of all regions are periodically multiplied with an aging factor (by default 0.8). This aging mechanism effectivelly reduces the relative importance of older accesses in favor of new accesses. Queries may cause records to be modified (item 6), resulting in pages marked "dirty" in the main memory bufferpool. Whenever a dirty page is displaced from the main memory bufferpool (item 5) it is written to the hard disk (HDD) and updated in the SCM (item 6), if present. Every time a page is read from either the SCM (item 2) or HDD (item 3), its identifier is also fed into the temperature-based replacement logic so that the temperature information is updated.

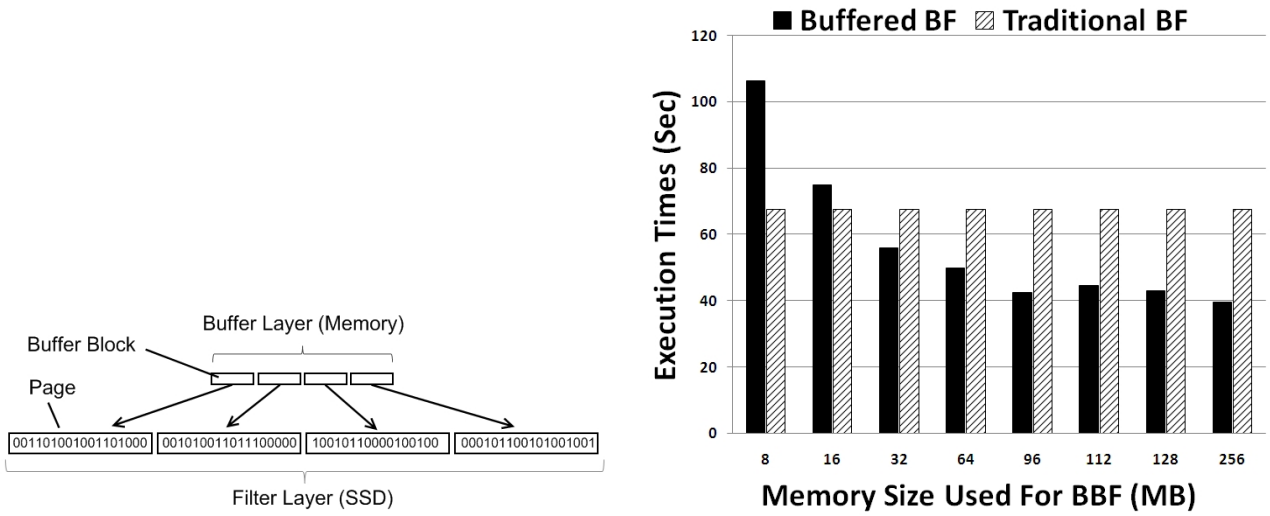
In order to compare the temperature-aware caching policy (TAC) with other replacement policies, we benchmarked them against a TPC-C [19] database (150 warehouses). Specifically, we compared TAC with the following page replacement policies: least recently used (LRU), first in first out (FIFO), clock replacement, and adaptive replacement (ARC) [9]. With this scaling factor the database occupies a total of 15 GB of disk space. For the memory bufferpool, 368 MB (2.5% of the database size) of space is used while running the transactions of 20 clients. The results for different ratios between the size of the SCM bufferpool and the main memory bufferpool are shown in Figure 1(b).

As the results prove, TAC outperforms the other replacement algorithms for all SCM bufferpool sizes. The difference in hit ratios is more pronounced for the case when the SCM bufferpool size is smaller (e.g. 1 or 2 times the size of the main memory bufferpool). This is due to the selective caching property of TAC (not caching pages colder than the coldest page cached), which reduces cache pollution. For large SCM bufferpool sizes, all algorithms (except for FIFO) perform about the same.

3 Extending Main Memory Data Structures using SCM

Besides the bufferpool, a database system employs main memory for a variety of other operations like locking, sorting, hashing, aggregations and bitmap operations. In this section, we show how the Bloom filter [2], can be redesigned to take advantage of a memory hierarchy consisting of main memory and SCM [3]. Database systems have used Bloom filters for index ANDing [1], join processing [10], selectivity estimation [20], and statistics collection [1, 20].

A traditional Bloom filter (TBF) consists of a vector of β bits, initially all set to 0. To update the filter,



(a) Structure

(b) Execution Times for Different Buffer Sizes

Figure 2: Buffered Bloom Filter.

k independent hash functions $h_1, h_2, ..., h_k$ all with range $\{1, ..., \beta\}$ are used. For each element $R \in S$, the bit locations $h_i(R)$, $i \in \{1, 2, ..., k\}$, are set to 1. To check if an element e is in S, the bit locations $h_i(e)$, $i \in \{1, 2, ..., k\}$, are checked. If any of them is not set to 1, we conclude that e is not an element of S. Otherwise, we assume that e is an element of S, with a certain probability that the assumption is wrong. If we conclude that the element is in the set although it is not, that element is called a false positive. The false positive rate is directly proportional to the compression ratio. Thus, in order to lower the false positive rate, a larger amount of memory is required.

We now present a Buffered Bloom Filter (BBF) algorithm that runs on a memory hierarchy consisting of DRAM and SCM. This algorithm reduces the main memory requirement and the false positive rate without any performance loss. Given that flash is one order of magnitude slower that main memory, a straight forward moving of the filter to flash will result in significant performance loss. To compensate for the performance loss, we defer the read and write operations during the execution of both build and probe phases. With this improvement, SSD page accesses are reduced by up to three orders of magnitude depending on the buffer size. We also divide the Bloom filter into virtual sub-filters to improve the locality of the bit operations. This reduces the CPU cache misses.

The hierarchical structure of the BBF is depicted in Figure 2(a). The upper layer (called the *buffer layer*) is stored in main memory while the lower layer (the *filter layer*) is stored in the SSD. The former is used to buffer the auxiliary information pertaining to the deferred reads and writes. The latter is used to store the filter's bit vector. The filter is divided into virtual pages such that each page keeps the bit vector of a sub-filter. If the filter is divided into δ pages, the buffered Bloom filter can be considered as a combination of δ separate traditional Bloom filters. Each sub-filter has a dedicated buffer block.

We experimentally examined the impact of buffer size on execution time of the BBF while specifying a upper bound of 0.25% on the false positive rate. For these experiments, the number of build and probe records was kept at 50 Million. We used 2 hash functions for the BBF with a filter size of 256 MB (on SSD) and page size of 2MB. For the TBF we used 3 hash functions with a filter size of 128 MB (in DRAM). The execution times of each run for different buffer sizes are given in Figure 2(b). The BBF runs faster than the TBF provided that the buffer size is at least 32 MB of memory space. We also measured the false positives in an experiment setting where we used 32 MB of memory space for BBF and 64 MB for TBF. We observed that TBF yields at least an order of magnitude more false positives than BBF does.

4 Conclusion

In this paper we have described our work in the exploitation of SCM for database systems. We presented some of our early results in placement of objects in storage and memory systems that have SCM in a hierarchy or in the same level with other storage devices. We also examined some of the challenges of adapting data structures which are inherently main memory based to work out of a memory hierarchy consisting of DRAM and SCM.

While our approaches have been validated using flash devices, similar principles would be applicable (with slightly different tuning parameters) for other SCM devices like Phase Change Memory. Certain potential features of PCM, such as byte-addressability and the absence of an erase cycle, may make PCM superior to NAND flash for some workloads.

References

[1] A. Balmin, T. Eliaz, J. Hornibrook, L. Lim, G. M. Lohman, D. E. Simmen, M. Wang, and C. Zhang. Cost-based optimization in DB2 XML. *IBM Systems Journal*, 45(2):299–320, 2006.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM*, 13(7):422–426, 1970.

[3] M. Canim, G. A. Mihaila, B. Bhattacharjee, C. A. Lang, and K. A. Ross. Buffered Bloom filters on solid state storage. In *First Intl. Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS*10)*, 2010.

[4] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. An object placement advisor for DB2 using solid state storage. *Proc. VLDB Endow.*, 2(2):1318–1329, 2009.

[5] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *PVLDB*, 3(2):1435–1446, 2010.

[6] B. Dageville and K. Dias. Oracle's self-tuning architecture and solutions. *IEEE Data Eng. Bull.*, 29(3):24–31, 2006.

[7] DB2 for Linux, UNIX and Windows. `www-01.ibm.com/software/data/db2/linux-unix-windows`.

[8] R. F. Freitas and W. W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4-5):439–448, 2008.

[9] N. Megiddo and D. S. Modha. Outperforming LRU with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[10] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Trans. Software Eng.*, 16(5):558–560, 1990.

[11] A technical overview of the Sun Oracle Exadata storage server and database machine. Internet Publication, September 2009. `www.oracle.com/us/solutions/datawarehousing/039572.pdf`.

[12] Oracle 11g database. `www.oracledatabase11g.com`.

[13] K. A. Ross. Modeling the performance of algorithms on flash memory devices. In *DaMoN*. ACM, 2008.

[14] IBM Smart Analytics System. `www-01.ibm.com/software/data/infosphere/smart-analytics-system/`.

[15] SolidDB product family. `www-01.ibm.com/software/data/soliddb/`.

[16] A. J. Storm, C. Garcia-Arellano, S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *VLDB*, pages 1081–1092. ACM, 2006.

[17] Teradata extreme performance appliance. `teradata.de/t/assets/0/206/280/e30a3614-7a93-450d-9506-6eb6e461d0a2.pdf`.

[18] Oracle TimesTen in-memory database. `www.oracle.com/us/products/database/timesten/index.html`.

[19] TPC-C, On-Line Transaction Processing Benchmark. `www.tpc.org/tpcc/`.

[20] W. Wang, H. Jiang, H. Lu, and J. X. Yu. Bloom histogram: Path selectivity estimation for xml data with updates. In *VLDB*, pages 240–251. 2004.