On Configuring a Hierarchy of Storage Media in the Age of NVM

Shahram Ghandeharizadeh
Database Laboratory
University of Southern California
Los Angeles, CA 90089
shahram@usc.edu

Sandy Irani
Computer Science Department
University of California, Irvine
Irvine, CA 92697
irani@ics.uci.edu

Jenny Lam
Department of Computer Science
San Jose State University
San Jose, CA 95192
jenny.lam01@sjsu.edu

***Abstract*—Advances in storage technology have introduced Non-Volatile Memory, NVM, as a new storage medium. NVM, along with DRAM and Disk present a system designer with a wide array of options in designing caching middleware. Moreover, design decisions to replicate a data item in more than one level of a caching memory hierarchy may enhance the overall system performance with a faster recovery time in the event of a memory failure. Given a fixed budget, the key configuration questions are: Which storage media should constitute the memory hierarchy? What is the storage capacity of each hierarchy? Should data be replicated or partitioned across the different levels of the hierarchy? We study a model of these cache configuration questions and present results from a simple algorithm to evaluate design tradeoffs in the context of a memory hierarchy for a Key-Value Store, e.g., memcached. The results show selective replication is appropriate with certain failure rates and workload characteristics. With a slim failure rate and frequent data updates, tiering of data across the different storage media that constitute the cache is superior to replication.**

I. Introduction

The storage industry has advanced to introduce Non-Volatile Memory (NVM) such as PCM, STT-RAM, and NAND Flash as new storage media. This new form of storage is anticipated to be much faster than Disk as permanent store and less expensive than DRAM as volatile memory. When compared with DRAM, NVM retains its content in the presence of power failures and provides performance that is significantly faster than today's disk [1], [2], [3], [4], [5], [6]. While some NVM such as Memristor [7] are anticipated to be byte-addressable, others such as NAND Flash are block-based.

In this paper, we propose a framework that uses the characteristics of an application database and its workload to address the challenge of how best to use this new technology for caching middleware. In particular, we study an application-side cache that augments a data store with a key-value store (KVS) and enables an application to lookup the result of queries issued repeatedly by performing random accesses. It is designed for workloads that exhibit a high read to write ratio such as social networking. An example KVS is memcached in use by Facebook that reports a 500 read to 1 write workload characteristic [8]. The extended version of the paper show how the model is applied to a host-side cache [9] [10].

In this paper, we use the term *cache* to refer to either one or several storage media used as a temporary staging area for data items. For example, in a KVS, the cache may consist of DRAM and NVM as two stashes.

We consider both *tiering* and *replication* of data items across stashes. Tiering, also termed *exclusive*, maintains only one copy of a data item across the stashes. Replication, also termed *inclusive*, constructs one or more copies of a data item across stashes. Replication can be costly if the data item is updated frequently. However, it expedites recovery of the data item in the presence of a stash failure, e.g., power failure with DRAM and hardware failure with NVM. We capture this tradeoff in the formulated optimization problem and its solution.

The *primary contribution* of this paper is two folds. First, an offline optimal algorithm that computes (1) the choice and sizes of the stashes that constitute a cache given a fixed budget and (2) a mechanism to determine when it is better to replicate an item across more than one stash. The algorithm simultaneously optimizes both questions subject to a fixed budgetary constraint.

The input to the algorithm is the workload of an application (frequency of access to its referenced data items, sizes of the data items, and the time to fetch a data item on a cache miss) and the characteristics of candidate stashes (read and write latency and transfer times, failure rate, and price). The proposed algorithm uses the distribution of access frequencies to guide overall design choices in determining how much if any of each type of storage media to use for the cache. The algorithm can also be used to guide high-level caching policy questions such as whether to maintain backup copies (replication) of data items in slower, more reliable storage media or whether to only keep a single copy of each item across stashes (tiering).

For example, Figure 1 shows the estimated average service time as a function of budget when the cache is limited to a single stash. Each line corresponds to a different storage medium used for the cache, with NVM_1 and NVM_2 corresponding to two representative NVM technologies (see Table I for their characteristics). With a tight budget (small x-axis values), NAND Flash is a better alternative than DRAM and comparable to NVM_1 because its inexpensive price facilitates a larger cache that enhances service time. Figure 1 illustrates that except in the extreme case where the budget is large enough to store the entire database in DRAM, the two

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00155

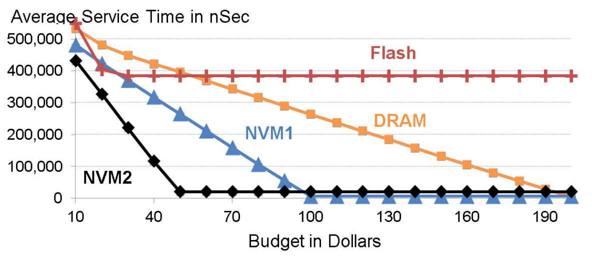

Fig. 1: Average service time of processing a social networking workload with different choice of storage medium for the cache.

NVM options are preferable storage media to DRAM. This information would influence a design choice in determining which type of memory to use to cache key-value pairs. The algorithm is flexible and performs the same optimization to realize caches consisting of two or more stashes. In addition, it evaluates choices in which data items are stored in more than one stash. Note that the algorithm uses the parameters of the storage technologies as input, so it can adapt to different types of storage media, depending on what is available.

A second contribution is our trace driven evaluation of the proposed method. The main lessons of this evaluation are:

1) Some combination of storage media perform considerably better than others over a wide range of budget constraints. For example, the combination of Flash and NVM_2 is a good choice in the context of host-side caches for most budget scenarios. DRAM and NVM_2 does best for key-value stores for most budgets.
2) After a certain threshold point (that depends on database size and workload characteristics), spending money on larger and faster caches does not offer significant improvement in performance.
3) Before spending money on a pricey NVM that stores a small fraction of data items, better performance gains might be achievable by purchasing a slightly slower speed storage medium with a higher storage capacity that can stage all data items.
4) Optional replication of data items by our algorithm produces results that are slightly better than tiering and significantly superior to an approach that replicates all data items across stashes.
5) There can be many very different placements that approximate the optimal placement in their average service time. These alternatives can be explored by limiting the set of placement options and comparing the average service time under the more restrictive scenario to the average service time in which all possibilities are allowed.

II. The Model and Algorithm

We model query sequences by a stream of independent events in which the occurrence of a particular query (key-value read request) or update (key-value write request) does not change the likelihood that a different query or update occurs in the near future. Independently generated events is the model employed by social networking benchmarks such as BG [11] and LinkBench[12]. If the probabilities of queries and updates to each key-value pair are known a priori, then the static assignment of key-value pairs to storage media that minimizes the expected time to satisfy the next request is provably optimal. In practice, the placement of data items to stashes would be periodically adjusted to reflect changing access patterns.

In our simulation studies, we estimate the probability that a particular key-value pair is requested by analyzing the frequency of requests to that key-value pair in the trace file. We then determine an optimal memory configuration based on those probabilities. Note that the outcome of the algorithm (the quantities and types of storage to buy in designing a cache) does not depend critically on the read and write frequencies of individual items as these items can be moved to different stashes over time. Instead, the outcome of the algorithm depends on the total size of items with similar read/write characteristics. The idea is to estimate the characteristics of the load using the collective history for the individual items.

The information that is needed for each key value pair k is $size(k)$ (the size of the key-value pair in bytes), $comp(k)$ (the time to compute the key-value pair from the database), $f_R(k)$ (the frequency of a read reference for key k), and $f_W(k)$, the frequency of a write reference for key k.

There is a set of $\mathcal{S}$ candidate stashes for the cache, each made of a different memory type. For example, Table I shows the memory device types[1] and their parameters used in our simulations. Hence, in our experimental studies for the KVS, $\mathcal{S}$ is defined as: $\mathcal{S} = \{\text{Disk}, \text{Flash}, NVM_2, NVM_1, \text{DRAM}\}$. The times to read or write k from s, ($T_R(s,k)$ and $T_W(s,k)$) are based on the size of k as well as the read/write latency and bandwidth for stash s. With block devices (e.g., Disk and Flash), we adjust the size of the key-value pair to be a multiple of block sizes, $\lceil \frac{size(k)}{blocksize} \rceil \times blocksize$. TW(s,k) is dependent on the write policy. It denotes the time to either write k in s (write-through and write-back) or delete k (write-around). Note that size-dependent latencies could easily be incorporated into the values $T_R(s,k)$ and $T_W(s,k)$, although we do not include that variant here.

A copy of a key-value pair k can be placed in one or more of the stashes. We define a *placement option* P for a key-value pair to be a subset of the set of stashes. For example, the placement option $P = \{Flash, DRAM\}$ represents having a copy of a key value pair on both Flash and DRAM. The placement option $\emptyset$ represents the scenario where a key-value pair is not stored in the cache at all. In each experiment, we define a set of possible placement options that allows us to study the trade-offs between different options. For example, in tiering, there is at most one copy of a key value pair in the entire cache, so the collection of possible placements would be

[1]Since parameters for emerging NVM technology are not completely known, we used two representative NVM types, which we call NVM_1 and NVM_2.

$\emptyset$, {Disk}, {Flash}, $\{NVM_2\}$, $\{NVM_1\}$, and {DRAM}. In examining the trade-off between tiering and replication for a system with Flash and DRAM, the set of possible placements would be $\emptyset$, {Flash}, {Disk}, and {Flash, Disk}. Placement is determined for each key-value pair, so the decision between tiering or replication is made on an item-by-item basis.

We define an optimization problem which minimizes the average service time of a request subject to an overall budget for the cache. A solution the problem is defined by a set of indicator variables $x_{P,k} \in \{0, 1\}$ for each key-value pair k and placement option P. If $P = \{Flash, DRAM\}$ and $x_{P,k} = 1$, then we are using replication with a copy of k on both the Flash stash and the DRAM stash. The constraint $\sum_P x_{P,k} = 1$ says that k has exactly one placement, where the sum is taken over all possible placement options, including $\emptyset$.

If $x_{P,k} = 1$, then we must purchase $size(k)$ bytes of memory for each stash in P. The total cost, summed over all key-value pairs, must be at most he overall budget M:

$$\sum_k \sum_P x_{P,k} \cdot \sum_{s:s\in P} size(k) \cdot costPerByte(s) \leq M.$$

For a key-value pair k and placement option P, we define $serv(P, k)$ as the average service time of requests referencing k if k is assigned to stashes specified by the placement P. The placement option $P = \emptyset$ is a special case because it does not assign k to the cache, requiring every read reference to compute k using the data store and incur its service time $comp(k)$. In this case, $serv(\emptyset, k) = f_R(k) \cdot comp(k)$.

For $P \neq \emptyset$, there are three components to the service time for a key value pair k if it is placed according to P: (1) the time spent reading k, (2) the time spent writing k, and (3) the average cost of restoring the copies of k to a failed stash after repair. We consider each in turn.

If key-value pair k is assigned according to P, then upon a read request to k, it is read from the stash with the fastest read time for k: $\Delta_R(P, k) = \min_{s\in P} T_R(s, k)$. The average service time to read k is its read frequency times the time for the read: $f_R(k)\Delta_R(P, k)$.

Upon a write to k, all the copies of k across the stashes dictated by P must be updated[2]: $\Delta_W(P, k) = \sum_{s\in P} T_W(s, k)$. The average time writing k will be the frequency of a write request to k times the time for the writes: $f_W(k)\Delta_W(P, k)$.

In the presence of failures, it may be advantageous to store a key-value pair k on more than one stash. If k is stored on two stashes and one fails, then the time to repopulate the failed stash is reduced by retrieving a copy of k from the other stash. This is the only motivation for replicating a key-value pair across more than one stash. However, having an extra copy of a key-value pair increases the cost of updating on writes. The optimization problem we define here automatically takes this tradeoff into account.

[2]An alternative definition is to assume concurrent writes to all copies with the slowest stash dictating the time to write k: $\Delta_W(P, k) = \max_{s\in P} T_R(s, k)$. A strength of the proposed model is its flexibility to include alternative definitions.

We define a failure event F as the set of stashes that fail at the same time. The rate of a particular failure event F, denoted by λ_F, is the reciprocal of the expected inter-arrival between two occurrences of F. The cost of a failure event is the cost of restoring the contents of each stash that has failed, including the cost to obtain each key-value pair k contained in one of the stashes in F from the fastest stash that is still available (or recomputing k from scratch in the event that there are no remaining copies in the cache) plus the cost of writing k to the recovered stashes that originally held a copy of k. The cost of failure recovery is folded into the expected service time $serv(P, k)$ for each possible key-value pair k and each placement option P. Note that we did not incorporate events in which more than one stash fails at a time into our simulation results because those events occur with a low probability. However, the model as described can handle multiple stash failures.

The goal is to select values for the variables $x_{P,k} \in \{0, 1\}$ that minimizes

$$\sum_P \sum_k x_{P,k} \cdot serv(P, k)$$

subject to the constraints that for each k, $\sum_P x_{P,k} = 1$, and $\sum_k \sum_P x_{P,k} \cdot \sum_{s:s\in P} size(k) \cdot costPerByte(s) \leq M$.

It can be shown that this optimization problem is equivalent to the Multiple Choice Knapsack Problem (MCKP), a special case of the Knapsack problem which is known to be NP-hard [13]. We consider a linear programming relaxation in which the indicator variables $x_{p,S}$ can be assigned real values in the range from 0 to 1 instead of $\{0, 1\}$ values. In the case of MCKP, the LP relaxation can be optimally solved by a simple greedy algorithm. We show that when the number of key-value pairs is much larger than the number of stashes, the number of fractionally assigned items is very small. Therefore the greedy algorithm gives an excellent approximation of the optimal solution to the original integral optimization problem.

III. Evaluation

We use our method for cache configuration as a tool to evaluate different mixes of stashes under varying budget constraints in the context of host-side caches and key-value stores. The results for host-side caches can be found in [9]. Table I shows the parameters for the five types of memory used in this study, including their read and write latency, read and write bandwidth, price in dollars per gigabyte and mean time between failures. The actual parameters of current NVM technology are still undetermined, so we selected two representative NVM types to use in the study. Our method can take as input any set of storage devices and corresponding parameters.

Today's key-value store caches such as memcached use DRAM to store key-value pairs. An instance loses its content in the presence of a power failure. In this section, we consider a memcached instance that might be configured with five possible memory types for the cache: Disk, Flash, NVM_2, NVM_1, and DRAM.

	NVM_1	NVM_2	DRAM	Flash	Disk
Read Latency in ns (δ_R)	30	70	10	25000	2×10^6
Write Latency in ns (δ_W)	95	500	10	2×10^5	2×10^6
Read Bandwidth in MB/sec (β_R)	10×1024	7×1024	10×1024	200	10
Write Bandwidth in MB/sec (β_W)	5×1024	1×1024	10×1024	100	10
Price in dollars per Gig	4	2	8	1	.1
MTTF/MTBF in hours	21875	43776	8750	87576	87576
MTTR in hours	24	24	10	24	24
MTTF/MTBF + MTTR in years	2.5	5	1	10	10

TABLE I: Parameter settings of storage medium used in experimental evaluation.

Our evaluation employs traces from a cache augmented SQL system that processes social networking actions issued by the BG benchmark [11]. The mix of actions is 99% read and 1% write which is typical of social networking sites such as Facebook [14]. The trace corresponds to approximately 40 minutes of requests in which there are 1.1 million requests to 564 thousand key-value pairs. The total size of the key-value pairs requested is slightly less than 25 gigabytes. The cost of storing the entire database on the most expensive stash, DRAM, is just under \$200. When a key-value pair is absent from the cache, it must be recomputed by issuing one or more queries to the SQL system after every read which references it. The time for this computation is provided in the trace file. An update (write request) to a key-value pair is an update to the relational data used to compute that key-value pair. If the key-value pair is not stored on a stash, it does not need to be refilled (written to the cache). The information that is extracted from the trace file for input to the cache design optimization problem is the size of each key-value pair, the time to compute the key-value pair, as well as the frequency with which that item is read and updated.

Figure 2 shows the optimal size of each stash under a tiering policy with all five memory types available as placement options. At each budget point, the vast majority of the key-value pairs were stored in three consecutive stashes which means that it was generally more cost-effective to clear out key-value pairs from very slow stashes before investing in much faster space for the high-frequency items. Although it was slightly better to have the low frequency key-value pairs in NVM_1, the effect on the cost was almost negligible if they were included in DRAM instead. This illustrates that there can be many substantially different placements that are all close to the optimal in their average service time. These alternatives can be explored by limiting the set of placement options and comparing the average service time under the more restrictive scenario to the average service time in which all possibilities are allowed.

We refer the interested reader to [9] for a complete description of the off-line algorithm and its evaluation for host-side caches.

Acknowledgment

Sandy Irani and Jenny Lam are supported in part by the NSF grant CCF-0916181. Jenny Lam is now at Google.

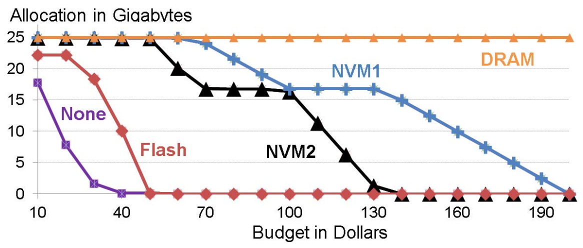

Fig. 2: The optimal partition of the key-value pairs among the stashes as the budget varies. The cost of stash failures is not included in the evaluation of optimality.

References

[1] J. Arulraj and A. Pavlo, "How to Build a Non-Volatile Memory Database Management System," in *SIGMOD*, 2017.

[2] X. Wu, F. Ni, L. Zhang, Y. Wang, Y. Ren, M. Hack, Z. Shao, and S. Jiang, "NVMcached: An NVM-based Key-Value Cache," in *ACM SIGOPS APSys*, 2016.

[3] J. Arulraj, M. Perron, and A. Pavlo, "Write-behind Logging," *VLDB*, vol. 10, no. 4, 2016.

[4] J. Ou, J. Shu, and Y. Lu, "A High Performance File System for Non-volatile Main Memory," in *EuroSys*, 2016.

[5] L. Sun, Y. Lu, and J. Shu, "DP2: Reducing Transaction Overhead with Differential and Dual Persistency in Persistent Memory," in *CF*, 2015.

[6] C. Shimin and J. Qin, "Persistent B+-Trees in Non-Volatile Main Memory," *PVLDB*, vol. 8, no. 7, 2015.

[7] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The Missing Memristor Found," *Nature*, vol. 7191, pp. 80–83, 2008.

[8] N. Bronson, T. Lento, and J. L. Wiener, "Open Data Challenges at Facebook," in *ICDE*, 2015, pp. 1516–1519.

[9] S. Ghandeharizadeh, S. Irani, and J. Lam, "On Configuring a Hierarchy of Storage Media in the Age of NVM," USC Database Laboratory, http://dblab.usc.edu/Users/papers/CacheDesTR2.pdf, Technical Report 2015-01, 2015.

[10] Y. Alabdulkarim, M. Almaymoni, Z. Cao, S. Ghandeharizadeh, H. Nguyen, and L. Song, "A Comparison of Flashcache with IQ-Twemcached," in *IEEE CloudDM*, 2016.

[11] S. Barahmand and S. Ghandeharizadeh, "BG: A Benchmark to Evaluate Interactive Social Networking Actions," *CIDR*, January 2013.

[12] T. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: A Database Benchmark Based on the Facebook Social Graph," *ACM SIGMOD*, June 2013.

[13] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[14] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's Distributed Data Store for the Social Graph," in *USENIX ATC 13*, San Jose, CA, 2013, pp. 49–60.