Memory Hierarchy Design for Caching Middleware in the Age of NVM *

Shahram Ghandeharizadeh, Sandy Irani, Jenny Lam

Database Laboratory Technical Report 2015-01

Computer Science Department, USC

Los Angeles, California 90089-0781

August 24, 2015

Abstract

Advances in storage technology have introduced Non-Volatile Memory, NVM, as a new storage medium. NVM, along with DRAM, NAND Flash, and Disk present a system designer with a wide array of options in designing caching middleware. This paper provides a systematic way to use knowledge about the frequencies of read and write requests to individual data items in order to determine the optimal cache configuration given a fixed budget. The approach introduced in this paper incorporates the characteristics of each type of memory to answer key design questions such as: how much will an increase in budget improve expected retrieval/update times? If it is desirable to limit the number of different types of memory in a cache, then which types of memory should be used for a particular budget and database size? When is it advantageous to store multiple copies of data objects in order to quickly recover from a memory failure? The cache configuration problem is modeled as an instance of the Multiple Choice Knapsack Problem (MCKP). Although MCKP is NP-complete, its linear programming relaxation is efficiently solvable and can be used to closely approximate the optimal solution. We use an algorithm for MCKP to evaluate design trade-offs in the context of a memory hierarchy for a Key-Value Store (e.g., memcached) as well as a host-side cache (e.g., Flashcache) to store disk pages. The results show selective replication is appropriate with certain failure rates. With a slim failure rate, tiering of data across the different storage media that constitute the cache is superior to replication.

1 Introduction

The storage industry has advanced to introduce Non-Volatile Memory (NVM) such as PCM, STT-RAM, and NAND Flash as new storage media (see Table 1). 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.

*Sandy Irani and Jenny Lam are with the University of California, Irvine. Their research is supported in part by the NSF grant CCF-0916181.

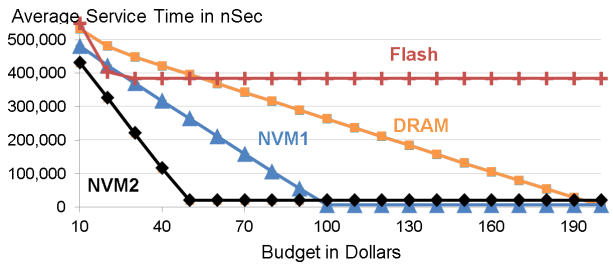


Figure 1: Average service time of processing a social networking workload with different choice of storage medium for the cache.

While some NVM such as Memristor [32] are anticipated to be byte-addressable, others such as NAND Flash are block-based.

Caching middleware is an immediate beneficiary of NVM. In this paper, we explore two possible extensions: either convert a volatile caching solution into a non-volatile one or use this faster storage for enhanced performance. We explore these two possibilities in the context of two different caching middleware. The first is cache augmented data store architecture [11] that extends a database management system with a key-value store (KVS) that enables an application to lookup the result of queries which are repeated many times. The key insight is that query result look up is faster and more efficient that query processing for those workloads that exhibit a high read to write ratio, e.g., social networking applications [27, 26, 11, 12]. An example is memcached in use by popular internet destinations such as Facebook [26]. It is a distributed in-memory KVS that augments a data store such as MySQL [1]. Today's memcached uses DRAM for fast storage and retrieval of key-value pairs. By using NVM as either a replacement[1] or extension of DRAM, one may convert memcached to retain its key-value pairs after a short-lived power failure. This raises the following research questions:

1. Given multiple choices of storage media and a fixed monetary budget, what combination of memory choices optimizes the performance of a workload? Is it appropriate to use only one storage medium or a mix?
2. What is the best policy for assigning key-value pairs across the selected choices of storage media? Should the system replicate key-value pairs across DRAM and NVM or partition them?

The second category of caching middleware are host-side caches that stage disk pages on a storage medium which is faster than permanent store and brings data closer to the application [28, 5, 30, 23, 31, 4, 13, 21, 17]. With these caches, a data item is either a disk block, a file, or an extent consisting of several blocks. For example, Dell Fluid Cache [5] stages disk pages referenced by a database management system such as MySQL on NAND Flash in order to hide the latency of retrieving these pages from a significantly slower permanent store. In our experiments with a social networking benchmark named BG, Fluid Cache enhanced the performance of MySQL anywhere from a factor of 3 to 20 [10]. It may be possible to boost performance further by extending this transparent cache to consist of both PCM and NAND Flash as PCM is significantly faster than NAND Flash. The same questions posed in the context of KVSs repeat themselves in the context of host-side caches.

[1]This requires the target NVM to be byte-addressable.

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. A *data item* might be a key-value pair with the KVS or a disk page with the host-side cache. Each type of storage medium that contributes space to the cache is termed a *stash*. For example, in a KVS, the cache may consist of DRAM and NVM as two stashes. The stashes in a host-side cache may be PCM and Flash. A copy of a data item occupying the cache is also stored on *permanent* store. With memcached, the permanent store is a data store such as MySQL from which the key-value pair must be recomputed. With a host-side cache such as Dell Fluid Cache, the permanent store may be a Storage Area Network (SAN) such as Dell Compellent. An application's read request for a data item fetches the data item from either the cache (a cache hit) or the permanent store (a cache miss) into *transient* memory for processing.

The different data items compete for the available space in the stashes that collectively constitute the cache. The goal is to assign data to the stashes with the objective of enhancing the performance of read and write operations referencing those data items. We consider both *tiering* and *replication* of data items across stashes. Tiering maintains only one copy of a data item across the stashes. Replication constructs one or more copies of a key-value pair across stashes.

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 static placement of data items across the stashes identified in Step 1. 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 bandwidths, 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).

While the algorithm presented here does compute a sample placement of data across stashes, the placement is primarily a tool for measuring the average response time for a particular choice of stash sizes. The actual placement of data to caches would naturally need to evolve in real time in order to adapt to fluctuations in the relative popularity of particular data items. This motivates the need for online algorithms for data placement as a future research direction, see Section 6. While a particular choice of stash sizes for a fixed budget may no longer be optimal if workload characteristics change, recent history is the best available source of information for future workload. The method proposed here provides a systematic way to use this information to guide choices that must be made at the time a system is configured.

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 3 for their characteristics). When the budget is tight (small values of x-axis), 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 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.

Our method for cache configuration specifies a size for each stash in bytes. However, one typically purchases memory in certain granularities. We assume that in determining the amount of memory for each stash, the byte figures would be rounded to the nearest value available for each memory type.

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

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 (See Figure 8). DRAM and NVM_2 does best for cache augmented data stores for most budgets (See Figure 12).
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 several 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.

The rest of this paper is organized as follows. Section 2 presents a model of the optimization problem using the language of cache augmented data stores as it is more general. Section 3 formalizes this optimization problem as an instance of the Multiple Choice Knapsack Problem and presents a near optimal algorithm to solve it. Section 4 demonstrates the effectiveness of this algorithm in deciding the choice of stashes and placement of data items across them for both cache augmented data stores and host-side caches using a trace-driven simulation study. We describe related work in Section 5 and detail brief future work in Section 6.

2 The Model

We describe the model using the language of key-value stores because it is the most general. At the end of Section 2, we describe the few changes to adapt our methods for designing a host-side cache.

	Memristor	FeRAM	PCM	STT-RAM	DRAM	NAND Flash	Disk
Read Time (ns)	< 10	20-40	20-70	10-30	10-50	25,000	2-8x10^6
Write Time (ns)	20-30	10-65	50-500	13-95	10-50	200,000	4-8x10^6
Retention	> 10 years	$\sim$10 years	< 10 years	Weeks	< 100 msec	$\sim$10 years	$\sim$10 years
Energy/bit (pJ)[2]	0.1-3	0.01-1	2-100	0.1-1	2-4	10-10^4	10^6-10^7
3D capability	Yes	Yes	Yes	No	No	Yes	N/A

Table 1: Alternative data storage technologies [7, 24].

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 [2] and LinkBench[1]. If the probabilities of queries and updates to each key-value pair are known a priori, then a static assignment of key-value pairs to each storage medium is optimal. Before each request, the optimal placement minimizes the expected time to satisfy the next request. If the distribution does not change over time, then the same placement will be optimal for every request. 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 the those probabilities.

For each key value pair k, we assume the following four quantities are known:

- $size(k)$ 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.
- $f_W(k)$ the frequency of a write reference for key k.

Note that $\sum_k (f_R(k) + f_W(k)) = 1$.

There is a set of $\mathcal{S}$ candidate stashes for the cache, each made of a different memory type. For example, Table 3 shows the memory device types[2] and their parameters used in our simulations. Hence, in our experimental studies for the KVS, $\mathcal{S}$ is defined as:

$$\mathcal{S} = \{\text{Disk}, \text{Flash}, NVM_2, NVM_1, \text{DRAM}\}.$$

Each stash $s \in \mathcal{S}$, has the following characteristics:

- $\delta_{R,s}$ the read latency of candidate stash s.
- $\delta_{W,s}$ the write latency of candidate stash s.
- $\beta_{R,s}$ the read bandwidth of candidate stash s.
- $\beta_{W,s}$ the write bandwidth of candidate stash s.
- $pricePerByte(s)$ the monetary cost of purchasing a byte of s.

The time to read k from s is:

$$T_R(s,k) = \delta_{R,s} + size(k)/\beta_{R,s}$$

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

The time to write k to s is:

$$T_W(s,k) = \delta_{W,s} + size(k)/\beta_{W,s}$$

2.1 Placement Options

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 for a key-value pairs 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:

$$\emptyset, \{\text{Disk}\}, \{\text{Flash}\}, \{NVM_2\}, \{NVM_1\}, \{\text{DRAM}\}.$$

In examining the trade-off between tiering and replication for a system with Flash and DRAM, the set of possible placements would be:

$$\emptyset, \{\text{Flash}\}, \{\text{Disk}\}, \{\text{Flash, Disk}\}.$$

We define an optimization problem that minimizes the expected time per request given a set of possible placement options and budget to purchase the memory for each stash. For each key-value pair k and each possible placement option P, there is an indicator variable $x_{P,k} \in \{0,1\}$, indicating whether k is placed according to 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 following constraint says that k has exactly one placement:

$$\sum_P x_{P,k} = 1,$$

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. Again, if $P = \{Flash, DRAM\}$, we need $size(k)$ bytes of Flash and $size(k)$ bytes of DRAM for the copies of key-value pair k. Thus, the monetary price of having key-value pair k in placement option P is:

$$price(P,k) = \sum_{s:s\in P} size(k) \cdot costPerByte(s).$$

If the overall budget is M, then the total cost, summed over all key-value pairs must be at most M:

$$\sum_k \sum_P x_{P,k} \cdot price(P,k) \leq M.$$

2.2 Expected Service Time

The objective of the optimization is to expedite the average processing time of a request. This translates to minimizing the average service time. 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 (i.e., the permanent 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. One may assume different models for the service time of this write operation such as a concurrent write to all copies with the slowest stash dictating the time to write ($\Delta_W(P, k) = \max_{s \in P} T_R(s, k)$) or a serial write where the different stashes with a copy are written one after another:

$$\Delta_W(P, k) = \sum_{s \in P} T_W(s, k).$$

The average time writing k will be the frequency of a write request to k times the time for the writes: $f_W(k)\Delta_W(P, k)$.

A strength of the proposed model is its flexibility to include alternative definitions of the write model. Our investigation of the concurrent and serial write models showed very little difference as the slowest stash dominates the write cost. Hence, the rest of this paper assumes a serial write model.

We model failures as a rate λ that defines the inter-arrival between two failures in terms of the number of requests as $\frac{1}{\lambda}$. For example, a failure rate of 0.001 (λ=0.001) means that the average number of requests between occurrences of two failures is 1000. We define a failure event F as the set of stashes that fail at the same time. To simplify discussion, we assume F consists of one failure. However, the model is general enough to express the optimization problem in which any possible subset F of stashes can fail. For each such failure event, we determine the cost of restoring the contents of the impacted stash. We require a failure rate λ_F for every possible failure event F. Section 4.1 details how we compute λ_F in our experiments using both trace files and parameter settings of stashes.

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. Having an extra copy of a key-value pair increases the cost of updating on writes. This trade-off between the cost of updating an additional copy and the benefit of having the extra copy in case of failure determines how many copies of a key-value pair is optimal. The optimization problem we define here automatically takes these considerations into account.

For each triple (F, P, k), we define $fail$, the cost of restoring k after a failure event F given that k is stored according to placement option P. $failCost(F, P, k) = 0$ if the set of stashes that fail has no overlap with P (i.e., $P \cap F = \emptyset$). Otherwise, there are two components to the cost: First, the cost of retrieving a copy of k. If all stashes of P are wiped clean after a failure event F (i.e. $P \subseteq F$), then $retrievalCost(F, P, k) = comp(k)$. Otherwise k is read from the fastest stash still available:

$$retrievalCost(F, P, k) = \min_{s \in P - F} \Delta_R(s, k).$$

The second component of the incurred cost after a failure is restoring k to the stashes in P that failed during failure event F:

$$restoreCost(F, P, k) = \sum_{s \in P \cap F} \Delta_W(s, k).$$

Now, we assemble all components of the service time of a request referencing a key-value pair k assigned according to the placement option P:

$$\begin{aligned} serv(P, k) &= f_R(k)\Delta_R(P, k) \\ &+ f_W(k)\Delta_W(P, k) \\ &+ \sum_F \lambda_F \cdot restoreCost(F, P, k) \\ &+ \sum_F \lambda_F \cdot retrievalCost(F, P, k) \end{aligned}$$

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

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

In the next section, we show how this optimization problem can be solved using known techniques for the Multiple Choice Knapsack Problem. Knapsack problems are typically maximization problems, so we define an equivalent maximization problem to the problem stated above which maximizes a benefit instead of minimizing a cost. For each placement P and key-value pair k, define:

$$ben(P, k) = serv(\emptyset, k) - serv(P, k).$$

Note that the benefit of placement $\emptyset$ is 0 (i.e., $ben(\emptyset, k) = 0$). The constraints and definitions for the problem are unchanged, except that the goal is now to select a placement for each k that maximizes:

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

An optimal solution to the minimization problem is also an optimal solution for the maximization problem and vice versa.

To summarize, the CACHE CONFIGURATION PROBLEM is to find a placement for each k (i.e., values for the indicator variables $x_{P,k} \in \{0, 1\}$) that maximizes:

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

subject to the constraints that for each k:

$$\sum_P x_{P,k} = 1,$$

and for budget M:

$$\sum_k \sum_P x_{P,k} \cdot price(P, k) \leq M.$$

Every sum over P is taken over the set of possible placement options that we define for a particular experiment (i.e., instance of the optimization problem). The values of the indicator variables dictate the size of the stashes: each stash must be large enough to hold a copy of every key-value pair that has a copy on that stash. For a stash s, we must sum over all placement options that include a copy of a key-value pair on s. The total size of stash s is then:

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

2.3 Host-Side Caches

We use the same framework to optimize a configuration for host-side caches. There are two key conceptual differences that are accommodated using our logical formalism. First, today's host-side caches uses Flash (instead of DRAM of cache augmented data stores, e.g., memcached) and it is natural to extend them with NVM. Hence, the set of stashes to consider are:

$$\mathcal{S} = \{\text{Flash}, NVM_1, NVM_2\}.$$

Second, a data item k might be either a disk page or a file. The cost of not assigning k to the cache means it must be serviced using the permanent store (might be a Disk controller, a RAID, a Storage Area Network):

$$serv(\emptyset, k) = f_R(k) \cdot T_R(Disk, k) + f_W(k) \cdot T_W(Disk, k).$$

$T_R(Disk, k)$ and $T_W(Disk, k)$ are the time to read and write k to Disk. Other definitions of Section 2 are unchanged.

3 The Multiple Choice Knapsack Problem

In the KNAPSACK PROBLEM, there is a set of items available to pack into a knapsack. Each item has a benefit and a weight. The knapsack has a weight limit and the goal is to select

the set of items to pack into the knapsack that maximizes the total benefit of the items selected while not exceeding the weight limit of the knapsack. In the Multiple Choice Knapsack Problem, the items are partitioned into groups with the additional constraint that at most one item from each group can be selected.

In the Cache Configuration Problem, each key-value pair has a set of placement options (including the option of not placing it on any of the memory banks). Therefore each key-value pair defines a class from which we are selecting at most one option. Each selection has an associated benefit (as defined in Section 2) and each selection has a price that corresponds to the "weight" of the choice in the knapsack problem. Therefore the Cache Configuration Problem can be cast as an instance of MCKP.

The Knapsack Problem is a well-studied problem in the theory literature and is known to be NP-hard [8]. Since the Knapsack problem is a special case of MCKP in which each item is in a category of its own, MCKP is also NP-hard. The books [22] and [16] are dedicated to the Knapsack Problem and its variants and both include a chapter on MCKP. One can 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 the following greedy algorithm: start with the placement $\emptyset$ for all of the key-value pairs. This placement has 0 overall benefit and 0 monetary cost. The algorithm considers a sequence of changes (to be defined later) in which the placement of a key-value pair is upgraded to a more beneficial and more costly placement option. This process continues until the money runs out and the last item can only be partially upgraded. The solution obtained by the greedy algorithm is optimal for the LP-relaxation of the problem [29] and therefore at least as good as the optimal integral solution. Moreover, only one item is fractionally placed. The algorithm used here follows the greedy algorithm but stops short of the last upgrade that results in a fractional placement. Let OPT_{frac} and OPT_{int} represent the total benefit obtained by the optimal solutions for the fractional and integer versions of MCKP respectively. Let GR_{frac} and GR_{int} represent the benefit obtained by the greedy algorithm for the fractional and integer versions of MCKP respectively. We have:

$$GR_{int} \leq OPT_{int} \leq OPT_{frac} = GR_{frac}.$$

Moreover, the difference in overall benefit between the greedy integral solution and the greedy fractional solution is at most the benefit obtained by the last fractional upgrade. Since individual key-value pairs are small with respect to the overall database size, the effect of not including the last partial upgrade is not significant. Thus, while MCKP is NP-complete, the method we use provably approximates the optimal cache configuration with additive error that is less than the benefit of a single key-value pair.

The running time of the algorithm depends on n, the number of key-value pairs, and p, the number of different placement options considered. With replication, the largest p would be is 2^m, where m is the number of different stashes since any subsets of the stashes could be a placement option. For tiering, $p = m + 1$ because each placement option is a single stash. The additional 1 comes from the $\emptyset$ option. Our implementation uses a priority queue to select the next upgrade resulting in an overall running time of $O(np \log np)$. There are more complex algorithms to find the greedy solution that run in time $O(np)$ [6, 34]. We chose the $O(np \log np)$ implementation because it was easier to implement and ran sufficiently quickly.

Recall that the algorithm is used as an offline step in the design of a cache, not to decide data placement in real time.

We now give a description of the greedy algorithm for MCKP. The p different placement options, $P_0, \ldots, P_{p-1}$, are sorted in increasing order by price, so $price(P_i, k) \leq price(P_{i+1}, k)$. The option P_0 is the $\emptyset$ option, and $price(P_0, k) = ben(P_0, k) = 0$.

Not every placement option is a reasonable choice for every key value pair. For example, if placement option P for key k has lower benefit and higher price than option P' for k, then P should not even be considered as a viable option for key k. Thus, the first step is a preprocessing step in which a list of viable options is determined for each k. If j is not on k's viable list, then P_j will never be considered as an option for k. The pseudo-code for SetViableOptions is given in Algorithm 1.

Algorithm 1 SetViableOptions(k).

```
Initialize viableList(k) to be an empty list
Initialize done = false
Initialize curr = 0
while (not done)
    viableList(k).add(curr)
    maxGradient = −∞
    for j = curr+1 ... p − 1
        deltaBen = ben(P_curr, k) − ben(P_j, k)
        deltaPrice = price(P_curr, k) − price(P_j, k)
        g = deltaBen/deltaPrice
        if (g > maxGradient)
            maxGradient = g
            next = j
    if (maxGradient > 0)
        curr = next
    else
        done = true
```

The procedure SetViableOptions is illustrated graphically in Figure 2. For this example, only two memory banks are considered: DRAM and FLASH. We consider four placement options for a key-value pair k. The key-value pair can be placed on both DRAM and FLASH (FD), Flash only (F), DRAM only (D), or neither ($\emptyset$). Each placement option is represented as a point with the horizontal axis representing the price of placing k on that placement option and the vertical axis representing the benefit. Essentially, a placement option is viable for k if its corresponding point lies on the convex hull of all the points and the slope of the segment connecting it to the previous point is positive.

The fact that the point corresponding to P_D is placed higher than P_{FD} for the particular key-value pair k used in Figure 2 means that $ben(P_D, k) > ben(P_{FD}, k)$. This arrangement may not be the same for all key-value pairs k and will in general, depend on a number of different parameters, especially the write frequency of k. The more frequently k is written, the more costly it is to maintain the extra copy of k on Flash. Figure 3 shows two other possible scenarios for the list of viable placements for a key-value pair.

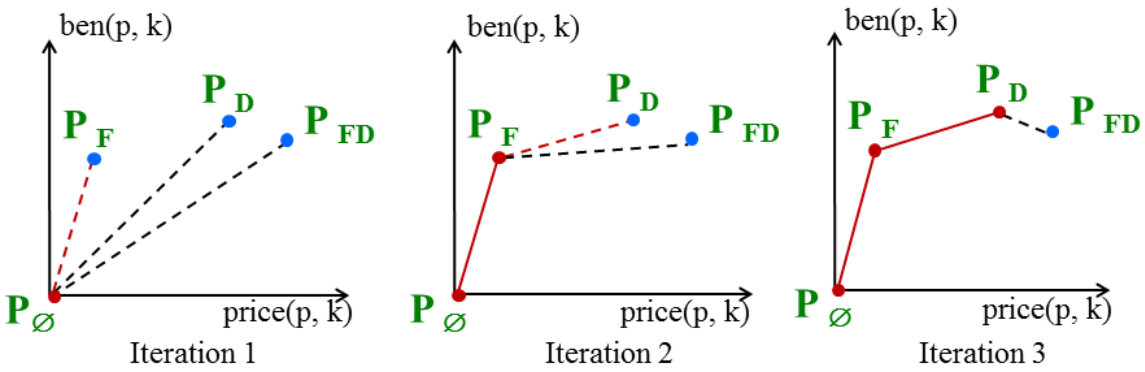


Figure 2: Illustration of SetViableOptions. Before the first iteration, the viableList for k is initialized to $[\emptyset]$. The algorithm examines each segment connecting $P_\emptyset$ to the three placements to the right and selects P_F because the segment from $P_\emptyset$ to P_F has the largest slope. The viableList for k is now $[\emptyset, \mathrm{F}]$. In the second iteration, the algorithm looks at the segments connecting P_F to the two placements to the right and selects P_D because the segment connecting P_F to P_D has the largest slope. The viableList for k is now $[\emptyset, \mathrm{F}, \mathrm{D}]$. FD is not added to k's viable list in the third iteration because the slope from P_D to P_{FD} is negative, indicating that placing k on Flash and DRAM costs more money and brings less benefit than placing k on DRAM only.

After the viable list for each key-value pair has been determined, the greedy algorithm initializes the placement for each key-value pair to $\emptyset$. In each iteration the greedy algorithm selects the key-value pair k such that the slope of the segment from k's current placement to k's next viable placement is maximized. Then greedy upgrades the placement for k to the next placement option on its list of viable placements. The process continues until the money runs out or until there is no upgrade that improves the overall benefit (i.e., until all the upgrade gradients are negative). The pseudo-code for the greedy algorithm is given in Algorithm 2 which makes use of a function that returns the upgrade gradient for a key value pair, given in Algorithm 3. The greedy algorithm is illustrated with a small example in Figure 4.

Example: We illustrate the GreedyPlacement algorithm with 3 key-value pairs using two candidate stashes DRAM and Flash. The graph for each key-value pair is shown in Figure 4. Each point in the graph is a placement option with the horizontal axis representing the price and the vertical axis representing benefit. The graph for k_1 and k_3 consists of two segments while the one for k_2 consists of three segments. The slope of a segment is shown by a value next to it. For example, the two segments of k_1 have a slope of 5 and 0.2, respectively.

The horizontal distance between the endpoints of the segments (distance between the dotted vertical lines) is the price of the upgrade. For example, with k_2, the price of an upgrade from FLASH to DRAM (distance between the vertical lines P_F and P_D) is higher than the upgrade price from DRAM to both FLASH and DRAM (distance between the vertical lines P_D and P_{FD}).

Table 2 shows the different iterations of the GreedyPlacement algorithm with a price of 1 for Flash and 4 for DRAM. We assume the size of the three key-value pairs is identical.

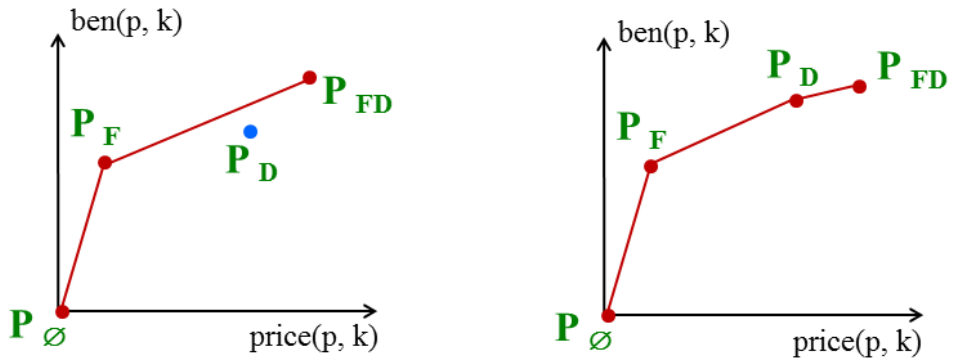


Figure 3: The viableList for the key on the left is $\langle \emptyset, F, FD \rangle$. After F was added, the segment from P_F to P_{FD} had a larger slope than the segment from P_F to P_D, so option D was bypassed and FD was added to k's viable list. The viableList for the key on the right is $\langle \emptyset, F, D, FD \rangle$.

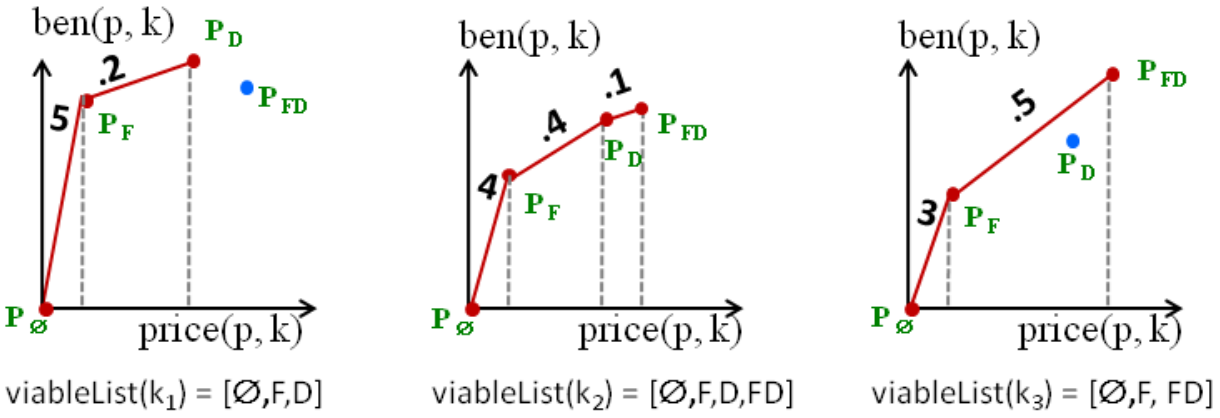


Figure 4: Three different key-value pairs and a graph of their placement options.

Table 2 shows how GreedyPlacement assigns according to segments with the highest slope first, see column 2 labeled "Segment Slope". The highest slope segment belongs to k_1, transitioning its assignment from $\emptyset$ to the placement option Flash. This is repeated with the second and third highest slope segments, transitioning the assignment of both k_2 and k_3 to Flash. The next highest slope segment is 0.5 (see row 4) and belongs to k_3. It corresponds to changing the placement from k_3 from DRAM to DRAM and Flash which entails replicating k_3 onto DRAM with an additional cost of 4, see last column of Table 2. The next line segment (slope of 0.4 belonging to k_2) changes the placement of k_2 from Flash to DRAM, resulting in a price of 3, i.e., cost of 4 for DRAM and saving of 1 by removing k_2 from Flash. This process continues until the available budget is exhausted.

A budget of 11 enables the first five iterations of GreedyPlacement. Its final placement will have k_1 on Flash, k_2 on Flash and DRAM, and k_3 on DRAM. A budget of 13 accommodates the sixth iteration to upgrade k_1 to DRAM. ■

Algorithm 2 GreedyPlacement(M as total budget)

```
Initialize moneySpent = 0
Initialize done = false
for all k
    viableList(k).reset()
while (not done)
    Select k with the largest upgrade gradient
    if GetUpgradeGradient(k) > 0
        next = viableList(k).getNext()
        current = viableList(k).getCurrent()
        deltaPrice = price(P_next, k) − price(P_curr, k)
        if deltaPrice + moneySpent ≤ M
            moneySpent = moneySpent + deltaPrice
            viableList(k).advance()
        else
            done = true
    else
        done = true
```

Algorithm 3 GetUpgradeGradient(k)

```
if viableList(k).isAtEnd()
    return −∞
next = viableList(k).getNext()
curr = viableList(k).getCurrent()
deltaBen = ben(P_next, k) − ben(P_curr, k)
deltaPrice = price(P_next, k) − price(P_curr, k)
return deltaBen/deltaPrice
```

Order	Segment Slope	Key Value Pair	Change in Placement	Upgrade Description	Price
1	5	k_1	$\emptyset \rightarrow$ F	Assign k_1 to Flash	1
2	4	k_2	$\emptyset \rightarrow$ F	Assign k_2 to Flash	1
3	3	k_3	$\emptyset \rightarrow$ F	Assign k_3 to Flash	1
4	.5	k_3	F $\rightarrow$ FD	Replicate k_3 to DRAM	4
5	.4	k_2	F $\rightarrow$ D	Move k_2 from Flash to DRAM	3
6	.2	k_1	F $\rightarrow$ D	Move k_1 from Flash to DRAM	3
7	.1	k_3	D $\rightarrow$ FD	Replicate k_3 to Flash	1

Table 2: Different iterations of the GreedyPlacement algorithm with the three keys. The order of upgrades is according to the slope of line segments shown in Figure 4.

4 Evaluation

In this section we used the algorithm presented in Section 3 as a tool to evaluate different mixes of stashes under varying budget constraints in the context of KVS and host-side caches. Table 3 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. The methods of Section 3 can take as input any set of storage devices and corresponding parameters.

A summary of the lessons learned from this evaluation are presented in Section 1. The rest of this section is organized as follows. In Subsection 4.1, we show how to compute the failure rates using the trace file in combination with the parameter settings of a candidate storage medium (see Table 3). Next, in Subsections 4.2 and 4.3 we present a trace-driven evaluation of host-side and KVS caches in turn.

4.1 Failure Rates

Recall from Section 2.2 that the inter-arrival between two failures is quantified in terms of the number of requests as $\frac{1}{\lambda}$ where λ is the rate of failures. For example, λ=0.001 means that on the average, there are 1000 requests between two failure occurrences. This models two kinds of failures: 1) power failures that cause a volatile stash such as DRAM to lose its content, and 2) hardware failures that require a stash such as NVM to be replaced with a new one. The former is characterized by Mean Time Between Failure (MTBF) and the latter is quantified using Mean Time To Failure (MTTF). Both model constant failure rates, meaning, in every time unit a failure has the same chance as any other time instance. MTBF is used for power failure because it pertains to a condition that is repairable. MTTF is used for devices because we assume they are non-repairable and must be replaced with a new one. Both incur a Mean Time To Repair (MTTR). With power failure, MTTR is the time to restore the power and for the system to warm-up the volatile memory with data. With NVM device failure, MTTR is the time for the system operator to shutdown the server, replace the failed NVM with a new one, restore the system to an operational state, and

	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 3: Parameter settings of storage medium used in experimental evaluation.

incur the overhead to populate the new empty NVM with data.

We require a failure rate λ_F for every possible failure event F. In our experiments, we only consider failure events that consist of a single device failure based on the assumption that failure events are sufficiently infrequent that it is unlikely that one memory device fails while another is still down. In these studies, to determine λ_F where F consists of a single stash (i.e., $F = \{s\}$), we multiply the number of requests per hour in the trace times ($MTTF+MTTR$) for non-volatile memories and ($MTBF + MTTR$) for DRAM. The number of requests per hour is determined by dividing the total number of requests in the trace by the time (in hours) over which the trace was gathered. A more elaborate way of modeling failure rates would be required for failure events in which more than one device fails.

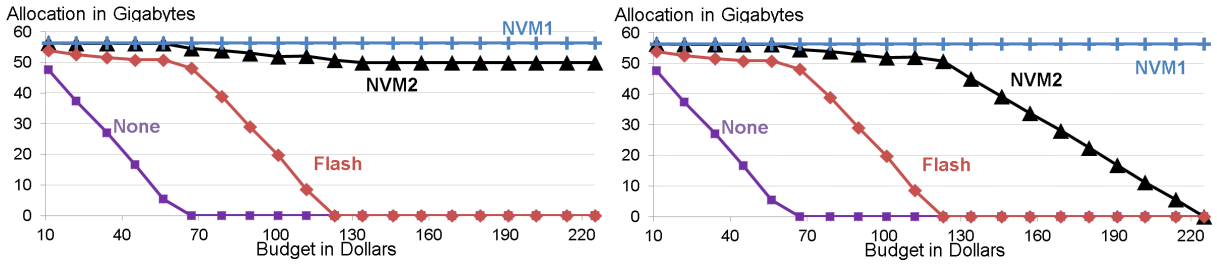
4.2 Host-Side Cache for Mail Server

Today's host-side caches employ one storage medium, namely Flash. This section considers an extension consisting of Flash, NVM_1 and NVM_2 as possible stashes. For our analysis, we used the disk block traces from a production mail server used by a University on a daily basis for one week [20] and several different production servers at Microsoft [15]. The latter includes the back-end server of Live Maps that displays satellite images and photographs of locations for an 18-hour period, the Microsoft Exchange server for an 18-hour weekday period, and a Micorosoft file server trace that covers a 6-hour period (see [15] for details). The former consists of 458 million requests to 14.7 million blocks. The total size of the requested blocks is 56.25 Gigabytes.

Obtained results from all traces are similar and highlight the following main lessons:

- The ability to replicate some objects (those that are not updated) does result in non-trivial improvement in average service time when failures are taken into account.
- Forcing replication is very costly - because the price is high for data that is updated.
- When a large budget is available, the optimal place for some objects is both stashes and for others is just the single fastest stash. This depends on whether they are updated.

Due to the similarity of the observed trends, this section presents experimental results from the University production mail server only. The first set of experiments examine different cache designs with a tiering policy in which each disk page is stored on at most one stash. In the first of these experiments, all three forms of memory (Flash, NVM_1 and NVM_2) are



(a) The cost of failures is included.

(b) The cost of failures is not included.

Figure 5: The optimal partition of the disk pages among the stashes as the budget varies. The amount allocated to each stash is the vertical distance between the line labeled with that memory type and the one below. In the first graph, the cost of failures is included. For budgets beyond \$124, most of the disk pages are in NVM_2. In the second graph, the cost of failures is not included. At the highest budget (\$225), all the disk pages are in NVM_1.

available as placement options. Figure 5a shows the allocation of blocks to stashes as the budget is increased up to \$225. For a particular budget, the size of each stash is the vertical distance between the line labeled with that stash and the line below. There is no budget at which the optimal allocation has disk pages in NVM_1 and disk pages not stored in the cache at all. In other words, before spending any money on upgrading disk blocks to NVM_1, it is more cost effective to get all of the disk blocks into the cache. There is sufficient variation in request frequency, however, that for budgets in the range of \$100, it is optimal to have disk pages spanning Flash, NVM_2, and NVM_1.

Another significant feature of the optimal allocation is that the allocation (and therefore the performance) does not change for budgets larger than \$124. Even though NVM_1 has both faster read and write times than NVM_2, it is more favorable to keep 89% of the disk pages in NVM_2, independent of cost. The reason is that failure costs play a significant role, despite the fact that the likelihood of a failure is very small. Disk blocks that are placed in NVM_2 under the optimal placement have relatively low request frequency (averaging 3.4 requests over the course of the week-long trace), and therefore, there is less advantage to having them in NVM_1. Even though failures are very unlikely, the difference in failure rate between NVM_1 and NVM_2 becomes significant in expectation when multiplied times the cost of recovering the page from Disk in the event of a failure. By contrast, the disk pages allocated to NVM_1 have a much higher request rate (averaging 233 over the course of the trace) and the benefit of the faster response time of NVM_1 more than offsets the increased probability of having to retrieve them from Disk in the case of a failure.

In a single server system, it may not make sense to average over the effect of events like failures that happen once every few years even if they are significant in expectation. On the other hand, with a multi-node cache configuration consisting of a large number of servers, the likelihood of failures in a shorter period of time is much higher and it is sensible to average in their impact.

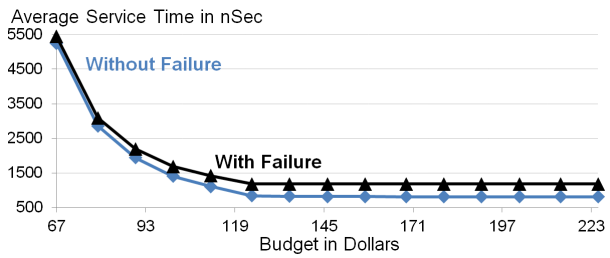


Figure 6: The average cost to service requests under the optimal cache configuration for different budgets.

Figure 5b shows the same experiment run except that the cost of failures is not included in the service time. The service time for a disk page is just the expected time servicing read and write requests to the page. With the cost of failures removed, the optimal placement (with unlimited budget) does have all of the disk pages in NVM_1. Although the allocation changes as the budget is increased, it's not clear whether the additional expenditure has a significant impact on the expected service time. Figure 6 shows the expected service time under the optimal configuration and placement for each budget. One line corresponds to the first set of allocations in which the cost of restoring after failures is taken into account. The second line corresponds to the scenario in which failures are not included. The difference between the two lines is the expected cost of failures. The difference is more pronounced (higher than 40% difference) with higher budgets as more disk pages are placed in either NVM_1 or NVM_2. The graph also indicates that performance does not improve significantly in either scenario past a budget of \$124. Thus, even though it is optimal with no failures to store everything in NVM_1 if the budget allows, the improvement past \$124 is not significant.

It may be favorable to have a cache consisting of fewer stashes. If so, our approach can be used to select a good set of memory types to include. In the next set of experiments we evaluate the impact on service time when the set of memory types is restricted. In Figure 7, each curve represents an experiment in which there is exactly one stash. A disk page can either be placed on that stash or excluded from the cache. NVM_2 generally does better than NVM_1 for anything but the full \$225 budget because more disk pages can fit in the cache. However, when the budget reaches the maximum \$225, the average response time with NVM_1 is 807 nS as compared to 3900 with NVM_2. In Figure 8, each curve represents an experiment in which there are exactly two stashes. A policy of tiering is employed, so a disk page can either be included in one of the two stashes or excluded from the cache. A combination of Flash and NVM_1 does better for a broader range of budgets, but NVM_2 and NVM_1 does better at the higher end of the scale.

Finally, we evaluate whether it is beneficial to allow some replication. Disk pages with low write rates will incur less overhead in having the additional copy on a less expensive but more reliable stash. Figure 9 is the allocation of disk pages to stashes when all 8 possible placements on the three stashes is allowed. No disk pages were ever placed on all three caches. The placement option $\{\text{Flash}, NVM_1\}$ had a very small allocation but was removed from the graph because it was too difficult to see.

Finally, we compare tiering and replication with a cache that includes Flash and NVM_1. We consider two variants of replication. Under *optional replication*, a key-value pair in

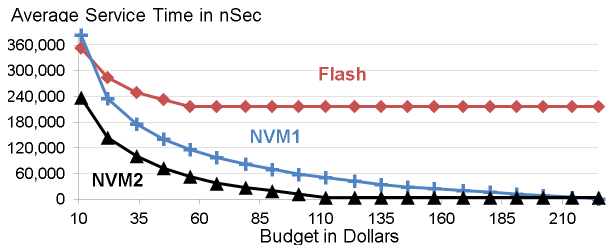


Figure 7: The average cost to service requests when there is only one stash. When the budget reaches the maximum \$225, the average response time with NVM_1 is 807 nS compared to 3900 with NVM_1

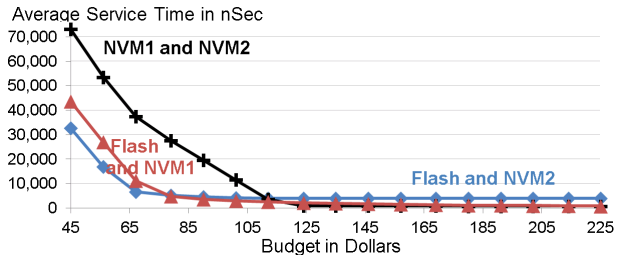


Figure 8: The average cost to service requests when there are two stashes. A tiering policy is employed so that a disk page an reside in at most on stash.

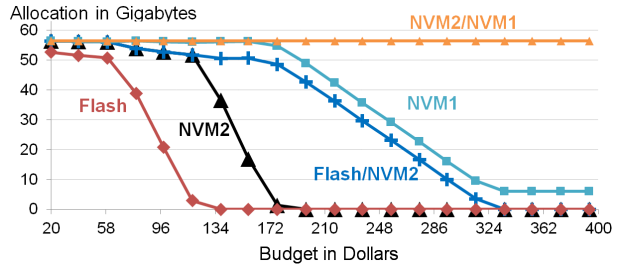


Figure 9: The optimal partition of disk pages among stashes when all 8 placements are allowed.

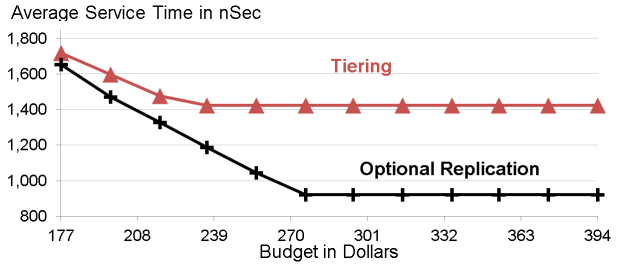


Figure 10: Tiering and optional replication with two stashes: Flash and NVM_1. The average service time for forced replication is not shown because even at the very highest budget range, the average response time was $212,459$ nS.

NVM_1 may or may not also reside in Flash. Under *forced replication*, every key-value pair in NVM_1 must also have a copy in Flash. Naturally, the most flexible variant (optional replication) will be at least as good as the other two policies (forced replication and tiering).

The graph in Figure 10 shows that the optimal placements under tiering and optional replication do differ as there are some disk pages that are written so infrequently that the cost of maintaining the additional copy is offset by the expected cost of restoring a copy to NVM_1 in case of a failure. The forced replication option is not even shown because it was very costly in comparison to the other two policies. The trace contained a significant number of disk pages that were updated frequently and therefore incur a high cost for replication. For the high budget range, the average response time for tiering and optional replication are approximately 1.4μs and .9μs, respectively. The corresponding value for forced replication is 212μs.

4.3 Cache-augmented Data Stores

Today's 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.

Our evaluation employs traces from a cache augmented SQL system that processes social networking actions issued by the BG benchmark [2]. The mix of actions is 99% read and 1% write which is typical of social networking sites such as Facebook [3]. 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).

In all of the budget scenarios considered, the optimal placement never assigned a key-value pair to Disk. For some key-value pairs, the cost of reading the key-value pair from Disk was more expensive than computing it directly from the database. Moreover, even for those key-value pairs which were were more expensive to compute than to retrieve from Disk, Disk was not a viable option because the corresponding point was not on the convex hull of placement options. (See the right graph of Figure 3 that illustrates a similar scenario in which DRAM is not an option for a key-value pair.)

Figure 11 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 not visible in the graph, under the optimal allocation, even for large budgets, there is approximately 8 MB of data that is not stored in the cache at all. These key-value pairs had one write request but no read requests over the

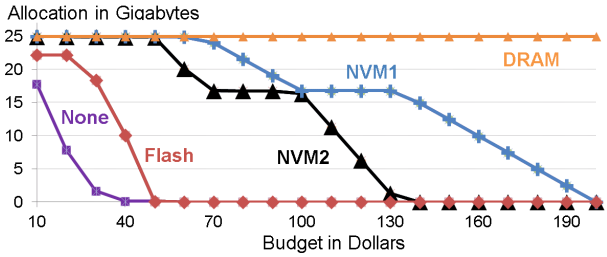


Figure 11: The optimal partition of the key-value pairs among the stashes as the budget varies. The cost of stash failures is not included in the evaluation of optimality.

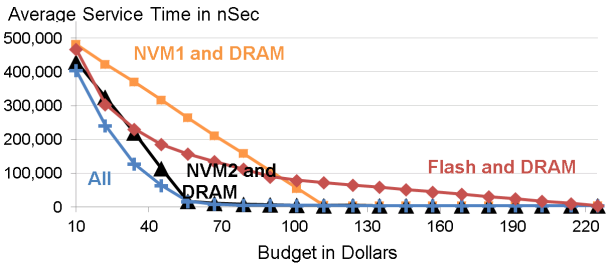


Figure 12: Comparison of cache configurations with two stashes with tiering.

course of the trace, so having those key-value pairs outside the cache reduced the average service time (although only slightly). The graph below shows the allocation in the scenario in which failure costs are not counted. When failures were counted, approximately 2/3 of the database was stored in NVM_1 instead of DRAM, even at the high end of the budget range. These key-value pairs were read only once during the entire trace in contrast with the key-value pairs stored in DRAM which were read on average about 4 times during the trace. 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. The next set of experiments carry out this idea.

We evaluate the cache performance when the cache consists of DRAM in combination with different types of NVM. Figure 12 shows the performance of the cache as a function of budget for the scenario where DRAM is combined with one other storage option. The combination that does well over the broadest range of budgets is NVM_2 and DRAM. NVM_1 and DRAM do the best at the highest price range, but NVM_2 and DRAM is very close. In Figure 1, where the cache is limited to one stash, NVM_2 provides the best service time with budgets lower than $100.

Finally, we compare tiering and replication with a cache that includes NVM_2 and DRAM. The graph in Figure 13 shows the data for tiering, optional replication and forced replication. Forced replication is significantly worse than the other two as the cost of updating key-value pairs in both stashes is expensive. The optimal placements under tiering and optional replication do differ slightly as the set of key-value pairs that are never updated are stored on

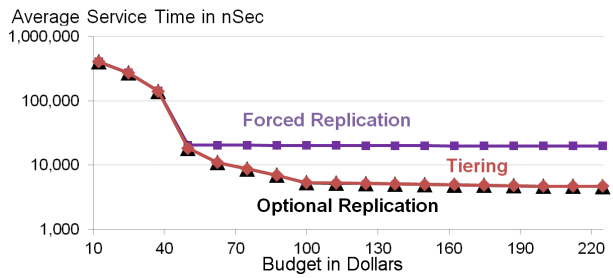


Figure 13: A comparison of tiering and replication policies with a cache that includes NVM_2 and DRAM. Under optional replication, a key-value pair can reside in NVM_2, DRAM or both. Under forced replication, every key-value pair in DRAM is also in NVM_2. In tiering, a key-value pair can only reside in one stash. The lines for Optional Replication and Tiering are extremely close.

both NVM_2 and DRAM under optional replication. However, the difference in performance is so negligible that the two lines cannot be distinguished in the graph. This data is more evidence that under this request distribution, the impact of memory failures is not significant and that tiering is a good choice for allocating key-value pairs to stashes.

5 Related Work

An overview of the different types of memory including NVM is provided in [14]. This study motivates the development of both offline and online algorithms for managing storage for database applications, but it does not present specific algorithms.

Several studies have investigated a multi-level cache hierarchy in the context of distributed file servers [25, 35]. These studies observe that LRU may not work well for the intermediate caches and present alternative online algorithms. The concept of *inclusive* and *exclusive* cache hierarchies is presented in [33]. Inclusive provides for duplication of disk blocks (similar to replication of data items) while exclusive de-duplicates blocks across the caches (tiering of data items). The approach in [33] is to extend LRU with a demote operation to implement an exclusive cache. None of these studies configure a cache by selecting the storage mediums that should participate as a stash in the multi-level hierarchy. Novel features of our proposed approach include its optimality in serving as a measuring yardstick to evaluate alternative on-line algorithms and its consideration of failure rates.

A cache hierarchy consisting of PCM and NAND Flash is analyzed in [18]. While the focus of this study is on PCM and its viability as a stash for use as a host-side cache, it presents an offline algorithm to tier 1 GB extents (consisting of 4 Kilobyte disk pages) across a hierarchy composed of PCM, Flash, and Disk. They evaluate the performance of a cache configuration in terms of I/Os per second (IOPS). Their method exhaustively searches all possible combinations of PCM, Flash, and Disk to find the one that maximizes (IOPS)/$. They use a heuristic to place data items into a candidate cache configuration in order to evaluate its performance (IOPS). We implemented their data placement algorithm and found that it did in fact find the optimal placement on the traces in our study. However, it is also possible to devise workload scenarios in which the data placement algorithm from is [18]

provably sub-optimal. An example is given in the appendix. Our approach is a superset of theirs as it considers both tiering and replication with an arbitrary mix of storage medium and failure rates. Moreover, our method simultaneously optimizes cache configuration and placement subject to a budget constraint. Finally, our method is provably optimal and can be used as a measuring yardstick to evaluate heuristics.

6 Future Work

An important next step is to evaluate online replacement policies in conjunction with the offline cache configuration method proposed here. This requires an extension of our model to consider the overhead of moving data items between the stashes. A simple approach may employ a static placement that is recomputed and updated periodically as the popularity of different items vary over time [19]. Alternatively, a cache replacement policy might be a variant of online algorithms for two-level hierarchies such as CAMP [9].

Another important direction to pursue is to evaluate how robust a cache design is to changes in the workload characteristics. The use of past statistics is bound to be only an approximation of the workload in the future. Therefore, it is important to understand how well a particular cache design does as the set of data items grows over time or as the characteristics of the access pattern change and when it is appropriate to alter the cache configuration.

References

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

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

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

[4] Byan, S., Lentini, J., Madan, A., Pabon, L., Condict, M., Kimmel, J., Kleiman, S., Small, C., and Storer, M. Mercury: Host-side Flash Caching for the Data Center. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)* (2012).

[5] Dell. Dell Fluid Cache for Storage Area Networks, http://www.dell.com/learn/us/en/04/solutions/fluid-cache-san, 2014.

[6] Dyer, M. An o(n) algorithm for the multiple-choice knapsack linear program. *Mathematical Programming 29*, 1 (1984), 57–63.

[7] Fink, M. Beyond DRAM and Flash, Part 2: New Memory Technology for the Data Deluge, HP Next, http://www8.hp.com/hpnext/posts/beyond-dram-and-flash-part-2-new-memory-technology-data-deluge.vcb6vrbcfe8, 2014.

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

[9] Ghandeharizadeh, S., Irani, S., Lam, J., and Yap, J. CAMP: A Cost Adaptive Multi-Queue Eviction Policy for Key-Value Stores. *Middleware* (2014).

[10] Ghandeharizadeh, S., Menon, J., Kotzur, G., Sen, S., and Chawla, G. Host Side Caching: Solutions and Opportunities. Technical Report 2015-01, USC Database Laboratory.

[11] Ghandeharizadeh, S., and Yap, J. Cache Augmented Database Management Systems. In *ACM SIGMOD DBSocial Workshop* (June 2013).

[12] Ghandeharizadeh, S., Yap, J., and Nguyen, H. Strong Consistency in Cache Augmented SQL Systems. *Middleware* (December 2014).

[13] Holland, D. A., Angelino, E., Wald, G., and Seltzer, M. I. Flash Caching on the Storage Client. In *USENIXATC* (2013).

[14] Hunter, H., Lastras-Montano, L., and Bhattacharjee, B. Adapting Server Systems for New Memory Technologies. *IEEE Computer 47*, 9 (Sept 2014), 78–84.

[15] Kavalanekar, S., Worthington, B. L., Zhang, Q., and Sharda, V. Characterization of Storage Workload Traces from Production Windows Servers. In *4th International Symposium on Workload Characterization IISWC, Seattle, Washington, USA, September* (2008), pp. 119–128.

[16] Kellerer, H., Pferschy, U., and Pisinger, D. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[17] Kim, H., Koltsidas, I., Ioannou, N., Seshadri, S., Muench, P., Dickey, C., and Chiu, L. Flash-Conscious Cache Population for Enterprise Database Workloads. In *Fifth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures* (2014).

[18] Kim, H., Seshadri, S., Dickey, C. L., and Chiu, L. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)* (2014).

[19] Kim, H., Seshadri, S., Dickey, C. L., and Chiu, L. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. In *USENIX FAST 14* (Santa Clara, CA, 2014), pp. 33–45.

[20] Koller, R., and Rangaswami, R. I/O deduplication: Utilizing content similarity to improve I/O performance. In *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010* (2010), pp. 211–224.

[21] Liu, D., Tai, J., Lo, J., Mi, N., and Zhu, X. VFRM: Flash Resource Manager in VMware ESX Server. In *IEEE Network Operations and Management Symposium* (2014).

[22] Martello, S., and Toth, P. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[23] Mituzas, D. Flashcache at Facebook: From 2010 to 2013 and Beyond, https://www.facebook.com/notes/facebook-engineering/flashcache-at-facebook-from-2010-to-2013-and-beyond/10151725297413920, 2010.

[24] Muller, C., Courtade, L., Turquat, C., Goux, L., and Wouters, D. Reliability of Three-Dimensional Ferroelectric Capacitor Memory-Like Arrays Simultane-

oulsy Submitted to X-Rays and Electrical Stresses. In *Non-Volatile Memory Technology Symposium* (2006).

[25] Muntz, D., and Honeyman, P. Multi-level caching in distributed file systems -or- your cache ain't nuthin' but trash. In *In Proceedings of the Winter 1992 USENIX* (1992), pp. 305–313.

[26] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling Memcache at Facebook. In *NSDI* (Berkeley, CA, 2013), USENIX, pp. 385–398.

[27] Ports, D. R. K., Clements, A. T., Zhang, I., Madden, S., and Liskov, B. Transactional Consistency and Automatic Management in an Application Data Cache. In *OSDI* (October 2010), USENIX.

[28] S. Daniel and S. Jafri. Using NetApp Flash Cache (PAM II) in Online Transaction Processing. NetApp White Paper, 2009.

[29] Sinha, P., and Zoltners, A. A. The multiple-choice knapsack problem. *Operations Research 27*, 3 (1979), pp. 503–515.

[30] Stearns, W., and Overstreet, K. Bcache: Caching Beyond Just RAM. https://lwn.net/Articles/394672/,http://bcache.evilpiepirate.org/, 2010.

[31] STEC. EnhanceIO SSD Caching Software, https://github.com/stec-inc/EnhanceIO, 2012.

[32] Strukov, D. B., Snider, G. S., Stewart, D. R., and Williams, R. S. The Missing Memristor Found. *Nature 7191* (2008), 80–83.

[33] Wong, T. M., and Wilkes, J. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC)* (2002), pp. 161–175.

[34] Zemel, E. An o(n) algorithm for the linear multiple choice knapsack problem and related problems. *Inf. Process. Lett. 18*, 3 (Mar. 1984), 123–128.

[35] Zhou, Y., Chen, Z., and Li, K. Second-Level Buffer Cache Management. *IEEE Trans. Parallel Distrib. Syst. 15*, 6 (June 2004).

Appendix

This section gives an example showing the non-optimality of the placement algorithm used in [18]. We use the same numbers for the device latencies as given in [18] and as written in the table below:

	PCM	Flash	15K Disk
T_R = 4 Ki R. Lat.	6.7 μs	108.0 μs	5000 μs
T_W = 4 Ki W. Lat.	128.3 μs	37.1 μs	5000 μs

The sample database is small with only two items. The whole example can be scaled by replicating each item n times and dividing all the frequencies by n. The space allocation is that there is enough PCM and Flash to each hold one of the two items, so neither will be stored on Disk. The frequencies are chosen so that the read rate is slightly higher than the write rate. The numbers can be altered to have different read-to-write ratios so that the

example will still hold.

$$\begin{aligned} f_R(1) &= 2.4N \\ f_W(1) &= 2N \\ f_R(2) &= 3.3N \\ f_W(2) &= N \end{aligned}$$

N is a normalizing constant so that the four values above all sum to 1. Since all the numbers for the rest of the example have a factor of N, we will drop the factor of N. The algorithm defines the following three values for each item:

$$\begin{aligned} Score_{PCM} &= f_R \cdot (T_R(Disk) - T_R(PCM)) \\ &+ f_W \cdot (T_W(Disk) - T_W(PCM)) \\ Score_{Flash} &= f_R \cdot (T_R(Disk) - T_R(Flash)) \\ &+ f_W \cdot (T_W(Disk) - T_W(Flash)) \\ Score &= \max\{Score_{PCM}, Score_{Flash}\} \end{aligned}$$

According to these definitions, both items prefer PCM:

$$\begin{aligned} Score_{PCM}(1) &= 2.4(5000 - 6.7) + 2(5000 - 128.3) \\ &= 21721 \\ Score_{Flash}(1) &= 2.4(5000 - 108) + 2(5000 - 37.1) \\ &= 21667 \\ Score(1) &= 21721 \end{aligned}$$

$$\begin{aligned} Score_{PCM}(2) &= 3.3(5000 - 6.7) + (5000 - 128.3) \\ &= 21350 \\ Score_{Flash}(2) &= 3.3(5000 - 108) + (5000 - 37.1) \\ &= 21107 \\ Score(2) &= 21350 \end{aligned}$$

The algorithm of [18] orders the items according to their score. The item with highest score (item 1) goes first and is placed in its first choice location (PCM). Then item 2 is placed in Flash. The overall expected time for an I/O is:

$$\begin{aligned} & f_R(1) \cdot ReadLat_{PCM} + f_W(1) \cdot WriteLat_{PCM} \\ + \; & f_R(2) \cdot ReadLat_{Flash} + f_W(2) \cdot WriteLat_{Flash} \\ = \; & 2.4 \cdot 6.7 + 2 * 128.3 + 3.3 * 108 + 37.1 \\ = \; & 666.18 \end{aligned}$$

According to the alternative placement, with item 1 in Flash and item 2 in PCM, the expected time for an I/O is:

$$\begin{aligned} & f_R(2) \cdot ReadLat_{PCM} + f_W(2) \cdot WriteLat_{PCM} \\ + \quad & f_R(1) \cdot ReadLat_{Flash} + f_W(1) \cdot WriteLat_{Flash} \\ = \quad & 3.3 \cdot 6.7 + 128.3 + 2.4 * 108 + 2 \cdot 37.1 \\ = \quad & 484.81 \end{aligned}$$

The placement not chosen by the algorithm has a significantly lower expected time per I/O.