RJ10517 (ALM1312-006) December 12, 2013
Computer Science

IBM Research Report

Cache Prediction for XIV

Ohad Rodeh, David Chambliss, Haim Helman
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Cache Prediction for XIV

Ohad Rodeh, David Chambliss, Haim Helman
IBM Almaden Research Center

May 22, 2014

Abstract

Enterprise block storage systems have been experiencing a revolution with the introduction of solid-state-disks, and the demise of high RPM disks. The IBM^{TM} XIV storage system in particular has embraced this concept with the introduction of large capacity SSD based caches. Customers can purchase the additional cache, for a price. The natural customer question is therefore, "will SSD caching improve the performance of my workload?".

To try to answer this question, and others like it, XIV has incorporated block-tracing into its platform, with complementary analysis and prediction tools.

We have constructed an efficient cache prediction tool, that works for a modern controller two level cache: RAM over SSD. It scales to terabytes of cache, and massive multi day traces. Given a customer workload, it can quickly figure out if the addition of SSD cache is likely to improve cache hit rates.

1 Introduction

Many scientific questions about the behavior of a block storage system cannot be answered using the succinct information that is normally available from its performance instrumentation. Timelines of aggregate IOs per second (IOps), throughput, latency, etc. are very good indicators of system health, adequacy of the performance, change in performance and utilization over time, and whether there are issues requiring deeper investigation. When a decision is faced regarding more sophisticated features, such as SSD caching, however, the statistical aggregates are not of much help. If we want to do better than applying rules of thumb, we need to capture more information about specific characteristics of workload patterns.

This work focuses on the collection and analysis of block traces. These are, essentially, very large tables containing a short descriptive record for each IO received by the storage target. No customer data is included in the IO record. Using these traces, we set out to solve an important customer question: "should I upgrade my XIV system with an SSD cache?"

The XIV storage system is an enterprise class storage controller constructed from a rack of modules, each holding a standard CPU, memory, and locally attached disks. It achieves very high throughput and reliability through its declustered distribution of all volumes across many disk drives. An important feature made available in the third generation (Gen3) of XIV is extended caching of read data on solid state disks (SSDs). The additional SSD cache may be 20× larger than the RAM cache. For workloads with the right access locality, SSD caching means that most read operations not already caught in the RAM cache will be SSD cache hits—meaning they are serviced with very low latency SSD reads instead of higher latency disk reads.

The XIV product offers a certain amount of SSD per module, for all modules. This means that, for a rack, a customer can buy a particular amount of flash cache. This limits customer choice, however, it simplifies testing and configuration. It also simplifies the prediction question. Instead of answering what performance is going be with varying amounts of SSD cache, we only need to answer that question for a particular amount of flash.

For many customers, adoption of XIV Gen3 has been contingent on the performance boost from SSD caching. Laboratory evaluations against meaningful benchmarks show large throughput increases and latency reductions when SSD caching is added to an XIV system [2]. However, some workloads do not benefit much from caching, and some customers have been reluctant to invest time and effort into a technology refresh without assurance that theirs is not one of those unlucky workloads. A primary goal of this work has been to offer such customers a reliable prediction on which to base their decisions. With this prediction in hand, a customer can make an informed choice, if he wants to spend the money and purchase the SSD.

We found that useful projections of cache effectiveness for a particular workload required more information than was captured routinely in aggregate performance statistics. Our choice was to capture traces of essentially all IOs to the storage system so that various paths for offline analysis could be tried. With an IO trace in hand, it is possible in principle to run the same algorithms as in the product and derive the correct cache results. We wanted a solution that would be fast, cheap to run, and be deployable with-

out making any changes to the XIV boxes in the field. Our solution was (1) trace the workload, (2) build a coarse cache simulation, and (3) run the trace through the simulator while sampling the IOs. We concluded that this was a workable compromise, that provided good accuracy while requiring little hardware resources.

There is wide existing literature on simulating cache and page re-reference behavior with LRU. We found that for realistic estimates, approximating cache behavior with LRU is insufficient, as it does not take into account optimizations for sequential access: read ahead, and write behind. We contribute:

1. A data structure that estimates re-references as well as sequentiality.
2. A sampling method that maintains memory overhead flat, at the cost of accuracy.
3. Real world experience with these methods.

We believe our methods are general, and can be applied to other workloads and systems.

This document is structured as follows: Section 2 describes related work, and Section 3 describes the controller. Section 4 describes our cache prediction procedure, Section 5 describes our field experiences, and Section 6 summarizes.

2 Related work

Seminal work [12] on demand paging introduced the key concept of a *working set*, and analyzed its theoretical behavior under simplifying statistical assumptions. The working set of a process $W(t, \tau)$, is the set of different pages accessed in the time interval $(t - \tau, t)$. The size of the working set, defined as $\omega(t, \tau)$, is very important for cache management and process scheduling. Stated simply, as long as the working set of a program is memory resident, thrashing will not occur.

Trace based LRU simulation has been extensively studied, for example [13] shows how to efficiently simulate multiple cache sizes in one trace pass.

The ESX server [4] uses a two level paging scheme, with guest OS page system layered on top of the hypervisor. It uses a combination of RAM page deduplication, swapping, and ballooning to manage memory. This supports over committing memory, while allowing efficient sharing, and reacting

quickly when memory pressure arises in a particular guest. The notion of working sets, and working set estimation is used to gauge idle memory.

Article [11] presents a method that, using efficient LRU simulation, allows estimating the Miss Ratio Curve (*MRC*) for a large number of cache sizes, in one trace run. Two practical schemes are presented for achieving this in a running Linux system, one is software only, the other employs hardware. The MRC shows for a process, given X amount of pages, how many cache misses it would suffer. This information allows splitting the available page cache between a group of processes, and reducing overall cache misses. Another use case, is to minimize the amount of RAM chips powered up, while maintaining good performance. This reduces energy usage.

The main difference between [11] and our work, is that we take into account not only LRU behavior, but also read-ahead, coalescing of dirty pages by the NVRAM, and additional XIV cache features. This complicates cache behavior significantly. It also means that if a page is in cache given X pages , it might not be in cache given $X + 1$ pages. Hence, we cannot calculate the MRC curve for multiple cache sizes in one trace run. In our use case, the flash size is known in advance, it is not a parameter.

RapidMRC [5] applies the MRC idea to the CPU L2 cache. The approach is to calculate the MRC online, inside the running machine. The main difficulties are (1) tracking L2 misses in software is expensive, and (2) the benefit from reducing L2 misses is not nearly as large as reducing page swaps to disk.

Synthetic traces have been used to evaluate systems and algorithms. For example [8] presents algorithms that build address traces that mimic locality of reference of executables. In the storage realm, the industry standard SPC1 and SPC2 benchmarks generate IO patterns that emulate behavior of real world applications.

IBM Redbook [2] provides a good description of XIV version 11.1. It describes the SSD caching, IO analytics, and visualization.

NetApp^{TM} Flash Cache [10] is a software/hardware offering that allows a customer to add flash cache to a NetApp^{TM} storage controller. A previous generation of the product was called *Performance Accelerated Module* (PAM), and it used DRAM instead of flash cache. At the time of writing, a single module can hold up to 1TB of SSD cache, and multiple modules can be connected to single NAS box. The product has a predictive mode where it simulates file-system cache accesses and returns a prediction for hit rates. This can be used by a customer considering buying more cache to see if the additional resources will benefit his workload.

In 2005 a storage simulator [1] was built for the massive storage system

used by the National Center for Atmospheric Research (NCAR). At the time, the total system capacity was 2PB. The simulator was trace based, it estimated overall system performance given a particular hardware configuration: disk drives, memory, tape systems etc. It worked by running a month worth of user read/write requests through a discrete-event based simulation. The goal was to help in capacity planning. For example, estimating how much disk cache is needed to offload reads from tapes. The authors report an accuracy level of approximately 20%.

Guerra et. al [6] describe a tiering algorithm that migrates extents dynamically between SAS, SATA, and SSD. Extents are significantly larger than filesystem pages, to reduce metadata costs. In the experiments, 64MB extents were used, with 200 bytes/extents bookkeeping overhead. The cost model takes into account IO density (IOps/GB), bandwidth, and cost per Gigabyte. Access and usage information is gathered per extent, and periodically, extent placement is determined by solving an optimization problem. By contrast, our work (1) uses caching, not tiering, so extent size is a 4KB page, (2) pages are moved online, as an immediate response to user requests, and (3) the cost model is fixed: the price of SSD for XIV is known to customers in advance.

Cache simulation using traces has a long academic tradition starting in the 1980s [7],[9]. Sampling has been used in combination with estimation of what would happen if cache was increased since the 1990s [3]. While the techniques exist, we have found that generic LRU simulation, frequency analysis, and re-access functions lead to high error rates. We had to simulate the particular XIV cache in order to achieve low error rates.

In addition, we had a wide body of real world traces to work with. These are, unfortunately, customer confidential, and cannot be freely shared.

3 XIV

3.1 System Overview

XIV is an enterprise storage system, supporting block level access using FCP and iSCSI protocols. Externally, it presents a single system image, with multiple FCP and Ethernet ports, which a customer can connect to his SAN. Data volumes can be created on the system, allowing a user to store data. XIV is typically used by higher end customers, for storing VM images, databases, health care information, ERP, etc.

The system comprises a rack of modules; see Figure 1. In the third hardware generation (Gen3), each module holds a server class CPU, 24 to

48 GB of RAM, and 12 SAS disks. An Infiniband network connects the modules, allowing fast low latency communication. Disks are not shared; each is connected to a single module. Systems with flash disks also have 512 to 800 GB of SSD per module, offering a total SSD capacity of 7.5 to 12 TB. The software running on Gen3 machines is XIV version 11. The second hardware generation (Gen2) used Ethernet for internal connectivity, 8 to 16 GB of RAM per module, and no SSDs. The software running on Gen2 machines is XIV version 10.

Figure 1: A XIV system with a full rack comprising 15 modules.

Customer data is striped across modules and disks in a declustered RAID-1 pattern. This ensures fault tolerance to single disk and module faults, while allowing fast rebuild. The slice-based data distribution places the responsibility for different megabyte units on different modules, in a manner that is rebuilt and rebalanced very rapidly when hardware is added, removed, or lost to a failure. Writes are replicated to two modules using NVRAM, improving write speed. Small IOs are coalesced in NVRAM, and written asynchronously to disk in large extents.

3.2 Caching

The controller includes hundreds of gigabytes of read-write RAM cache and a second, read-only, cache residing in terabytes of SSD. The SSD cache is

located at the lowest level of the IO stack, right above the disks. The cache for each module operates on its data only, independently from other modules. The cache is consulted prior to any disk read, and in most cases is populated immediately after a disk read or write. The cache design is focused on servicing small, random reads. It is engineered with write shaping and other flash-aware features to ensure high performance and good device lifetimes, even with lower-cost SSDs that do not themselves provide sustained high performance for random writes. Eviction decisions recognize groups of pages that are likely to reside on the same flash erase blocks. Another feature is that large and sequential IOs do not get populated into the cache. Several goals are met by this design:

1. Avoiding early SSD wear out
2. Making good use of high disk throughput
3. Reducing disk seeks

3.3 Trace Collection

The trace collection facility used in this study is built upon circular buffers of IO operation history to support engineering diagnostics. Trace collection operates in an external system that repeatedly polls the XIV modules to efficiently retrieve the new content from the buffers. The collection system processes the records from multiple nodes into a unified time-coherent log of all operations. Each operation record has seven fields:

num_blocks	number of 512-byte sectors
is_read	is this a read or a write request?
LBA	Logical Block Address
time	time when the request completed, in microseconds
latency	latency in microseconds
volume	volume ID as a 64-bit number
initiator_id	An identifier for the source of the I/O request

The collection facility also gathers configuration information which includes a volume table, an initiator list, and an aggregated performance statistics table. The volume table has per volume information like name, capacity, group, owner, and snapshot metadata. The initiator list holds the set of initiators connected to the storage target. The statistics table contains average latency, iops, and hit-rates, per five minute time slot. This is crucial for cache hit rate prediction.

The trace collection procedure has no discernible impact on the XIV system performance, which was essential for applying this to customers' production systems. It normally succeeds at collecting 100% of the IO records. Some records might be dropped if the collection process faces serious network congestion or scheduling congestion on its host machine. In early stages of this work we explored collection-time sampling to reduce data size, for example by collecting for two seconds out of every 10-second window. Such sampled data can yield some useful results, but we found in most cases it cannot provide the accuracy or confidence for cache prediction that we require. Thus it was necessary to optimize the procedure to achieve 100% collection without performance impact.

4 Cache prediction

This section presents the cache prediction method we developed and evaluated. Success in this modeling is a challenge because the hundreds of millions of IOs may all reference distinct 4KB pages, and a perfect model must potentially detect at each step a re-reference to any prior page. In addition, the cache sizes we consider have very long residency times, a hit can occur to data that is a week old. This requires keeping track of long histories.

XIV allows two configurations: all modules are fitted with SSD, or none of them are. Per module, one SSD chip of fixed size is installed. Our customers, those with existing XIV racks, can either buy SSD for all modules or none. This reduces the prediction question to a simulation for one total cache amount per rack. On the one hand, this is simpler than [13], where multiple cache sizes can be simulated in one trace run. On the other hand, the XIV write coalescing and prefetch algorithms, while providing substantial performance benefits, do not respect the *inclusion property*:

> If a page is in cache with K pages, it will be in a cache with $K + 1$ pages.

We suspect this is true for other real world cache implementations. Without the inclusion property, the stack model [12],[13], [11] breaks. For example, if an application does a sequential scan of a database, it will incur 100% cache misses, using the pure LRU stack model. By contrast, the XIV readahead algorithms will recognize the access pattern, prefetch the data from disk into RAM, and achieve close to 100% cache hits.

4.1 Objectives and requirements

The primary goal of modeling in this work is to provide what-if predictions of the performance effects from adding SSD cache to an existing XIV system, on the particular workloads of interest. The results should be fast enough and accurate enough to support good decisions about those changes. The modeling should not require unusual or expensive hardware, and preferably could be run on inexpensive laptops, so that it can be included in tools readily deployed for use in the field.

The customer's main concern is with the storage system's effect on application performance, and the critical factor is the IO latency. Since reads from SSDs are so much faster than disk reads, what matters most is what fraction of reads are serviced from the SSD cache: the hit rate. In some cases, it is important to translate the hit-rate change into an estimate of IO latency improvement.

The scope of prediction requirements is limited. Since decisions are based on mainstream cases and a normal operating state, the model does not need to handle complexities such as disk failure and performance during recovery. For this work, modeling of disk queues and consequent performance, and any explicitly time-dependent behavior, was not required. The one crucial result is the hit rate that results from caching, for the system as a whole and for individual volumes. Accurately identifying whether each individual read would be a hit or a miss is not required, though there is added value when hit-miss statistics can be projected to units smaller than entire volumes. We determined that an accuracy of 5% on system- and volume-level hit rates with SSDs would be sufficient for decisions. This is small compared to typical increases in hit rates with addition of SSD cache, which range from 10% (in systems where the RAM cache is already highly effective and there is little room for improvement) to more than 60%.

Another simplification is that our accuracy requirements for RAM cache modeling are relaxed. In what-if scenarios with SSDs, it is not very important which hits occur in RAM and which in SSD. For actual systems with RAM only, the hit rates are directly measured so model results need not be relied on. We would need better accuracy if customers were contemplating the removal of SSDs already deployed, and we were providing what-if analysis, but that need has not arisen and seems unlikely.

Approximate methods are required because of the cache sizes and the number of IOs. A straightforward implementation of a 6TB page-level cache would require 12GB to 24GB for metadata alone, far beyond our target footprint. Thus it is important that perfect accuracy is not required.

4.2 Overview

The cache prediction model is simulation based. It is written in Java^{TM}, uses one thread of execution and up to 2GB of RAM, and as such is fairly portable. XIV looks at 1MB on-disk areas, aligned on 1MB boundaries, this is mimicked by the cache model.

Several simplifying assumptions are made, the major ones are:

1. The cache and SSD are managed as a single global pool. In reality, these resources are split into 15 modules, spread across the rack. We implicitly assume an even distribution of IOs across the modules.

2. The caching algorithm is a single LRU list. The reality is that there is a cache instance per module, and several LRU lists are used to manage it.

3. Disk and module failures can be ignored.

4. Rebuild overheads can be ignored.

5. Readahead and write coalescing can be simplified.

The simulation makes a single pass on the trace, and performs bookkeeping for each IO. It also does bookkeeping every minute of simulated time. The running time is determined by the number of IOs in the trace, and the number of minutes simulated.

The address space is split into 16MB areas, and controller cache behavior is emulated for each region separately. Each region includes 4096 pages of size 4KB. Per page, 4 bytes and one bit are used, as follows:

Size	**Name**	**Usage**
2bytes	ramAge	how recently was this page touched in RAM
1bit	dirty	was this page written to in the last minute?
2bytes	ssdAge	how recently was this page touched in SSD

The RAM simulation works as follows. An untouched page has a zero *ramAge* counter. The first time page P is accessed, it's ramAge counter is set to MAX_SHORT ($2^{15} - 1$). The counter is reduced every minute by one. Once a minute, all regions are scanned, and the current occupancy is calculated. If the occupancy goes above the simulated cache size, the eviction routine is invoked. It discards clean pages by setting their ramAge to zero. Dirty pages cannot simply be discarded, they must be written to disk first (more on that below). The eviction routine first discards all pages

aged 1, if that is insufficient, it discards all pages aged 2, this continues until we fit into the amount of simulated cache.

It is theoretically possible for a page to live for MAX_SHORT minutes and then be discarded even if that is not strictly needed. However, this would require a trace lasting more than eight days, where our longest trace is a week.

Sampling is used to limit the number of simulated regions, this is explained in depth in 4.4. The default heap size is 2GB, which allows roughly 20,000 regions.

In theory, about four bytes of metadata are needed to keep track of a cache page. In practice, due to implementation overheads, and garbage collection costs, 8 bytes is a more realistic estimate. In order to simulate a gigabyte of cache, $8bytes \times \frac{1GB}{4KB} = 2MB$ of RAM are needed. With a memory budget of 2GB, a total of 1TB of cache can be simulated. Since we are trying to simulate a 7TB cache size, and a 243TB address space, we are likely to run out of RAM at some point. Sampling is used to overcome this problem, however, we need to account for the missing regions. This is done by *inflating* the values for the regions that are being simulated. For example, in order to estimate total IOps, assuming a sampling of 10%, we need to multiply the observed iops by a factor of 10. A more important example, is accounting for cache space. If the total amount of simulated cache is 500GB, then the actual amount is 5TB.

There are two kinds of prefetch: *short*, and *long*. The minute-scan looks for consecutive pages in a region with a maximal counter, these are extents read in the last minute. Long prefetch reads from disk 6MB, if the previous 3MB were read. Short prefetch works by extending read misses, and sending longer IOs to disk. Data prefetched in this way is brought into cache with a short counter, so that it will be evicted sooner, unless actually touched. The original read request R is extended in proportion to the size of the contiguous cache resident extent E, where $R \subseteq E$. The minimal IO sent to disk is 64KB, the maximal is 1MB, and in between $|E|$ is rounded down to the nearest power of two. For example, examine 1MB area that has been untouched so far, and the user is going to read it sequentially with 4KB IOs. The first IO $[0 - 4KB)$ is a miss, causing the range $[0 - 64KB)$ to be fetched from disk. The next 15 IOs are hits. The next IO, $[64KB - 68KB)$ is a miss, causing an additional 64KB to be fetched from disk ($[64KB - 128KB)$). Now come 15 page IOs that are hits, followed by a miss, causing a read of 128KB ($[128KB - 256KB)$). The next disk IOs are going to be $[256KB - 512KB)$, and $[512KB - 1024KB)$. In all, five disk IOs are used to serve 256 consecutive page read requests. After Three megabytes are read

in the way, long prefetch will kick in. In the XIV accounting method, this counts as five misses, and 251 hits.

To emulate the way XIV write caching works, the minute-scan finds dirty (written) extents. These are consecutive pages whose dirty bits are set. Such extents are written to disk and evicted at the minute boundary with IOs of up to 1MB. There is a limited budget of disk IOps, as a function of the number of backend disks. If the machine cannot keep up, then RAM resources will be taken up by dirty data that is waiting for destage. This causes a reduction in read hits, when there are write spikes.

The SSD cache is a layer situated between the RAM cache and disk. The SSD cache simulation uses a 2 byte counter for each accessed page, using an approach similar to RAM. The main interaction channel between RAM, SSD, and disk layers is through IO requests. All reads that miss the RAM cache are sent to the layer below it (SSD). All reads longer than 64KB bypass the SSD entirely, and go directly to disk. Short reads (up to 64KB) go to the SSD, if this is a hit, the data is copied from SSD to RAM. Otherwise, the read request continues to the disk, and populates both SSD and RAM on the way back. Write IOs are issued by the dirty-scan. All writes above 64KB invalidate the corresponding SSD area, on the way to the disk. All short writes update the SSD cache and the disk. Long IOs bypass the SSD in order to avoid wear out. This is important for workloads that have a significant sequential component.

For example, a 1MB write will invalidate the SSD cache, whereas a 32KB write will update it.

It would have been possible to use LRU to implement the cache, instead of page counters packed into regions. We believe it is much more efficient to use arrays of packed counters, than deal with an LRU stack, especially when there is a requirement to detect contiguous extents. There is one significant drawback to this approach, if data accesses are very sparse, we could end up, in the worst case, with a single valid page per region. That would be much less efficient than an LRU representation.

4.3 The traces

Table 4.3 shows summary information for a representative set of traces taken from our collection. Most workloads are around 24 hours, and represent real customer workloads. A few traces are shorter or longer, the shortest being four hours, and the longest 52 hours. There are a few benchmarks, notably EPIC, which is a health care database application, and SPC1, which is a storage benchmark. The number of IOs per trace ranges between hundred

millions, to several billion.

4.4 Sampling

Sampling is used to process fewer IOs and—more important—to contain the memory footprint. We use *space sampling* to reduce the number of tracked regions. A systematic subset of regions is selected, and all IOs and pages for those regions are processed, while IOs to regions outside the subset are ignored. The cache sizes applied to the subset are scaled down from the sizes in the full system by the sampling factor (i.e., the subset size divided by the total address space size).

We have also explored *time sampling* which might, for example, process only the first ten seconds of every minute of trace. If successful, time sampling would have empowered us to perform more streamlined data collection. The obvious pitfall is that processing some but not all IOs to a given address changes the correlations that lead to cache hits and changes the hit fraction, and there is no clear way to correct the distortion. Even without intentional time sampling, our procedures must deal with the possibility of occasional gaps in data collection, which affects accuracy.

Consider a 4KB page that belongs to region R, that is in the sample. The page was accessed ten times in one minute, with a 0.9 hit rate. Assume that trace collection missed one of these accesses, as it managed to collect 90% of the IOs. The simulation would count the accesses as one miss, to load the page from disk, and eight hits. This would mean $\frac{8}{9} = 0.89$ hit rate; a small error. However, if only 50% of the IOs are collected, the hit rate under simulation would be $\frac{4}{5} = 0.8$, a much larger error. The conclusion is that small IO collection losses are acceptable, but large loses cause significant simulation errors.

In our workloads, sometimes, a few regions account for a significant portion of the IO stream. Ignoring these hot spots, and sampling the regions uniformly, causes significant errors. Instead, we make an initial pass on the log, and count how many IOs land on each region. The regions are sorted from top to bottom by the number of IOs. The top 100 are chosen, and the rest are randomly selected, such that we are able to fill the available RAM. In what follows we call the top 100, *hot regions*, and the rest *cold regions*.

A hot region represents itself, while a cold region represents a group of regions. For example, if 100,000 regions were accessed in the trace, and 20,000 fit in memory, then the sampling factor is 5, and the *sample ratio* is $\frac{1}{5}$. A cold region in the sample represents 5 regions. An IO to a hot region is tallied as a single IO, while an IO to a cold region counts as 5.

NumIOs	runtime hours	description
892,421,450	12	SQL Server and VMware workloads
504,228,777	24	Business Intelligence (BI) on a Microsoft SQL cluster
761,785,357	24	VMware and Mysql database
1,937,988,132	24	SAP DB running on Microsoft SQL 2008 R2
714,679,893	24	Federated SQL database
705,010,671	24	Exchange workload
899,818,542	24	DB2 and several SQL servsers and VMware clients
918,274,570	24	Database server doing Oracle ERP (BI) work
693,292,344	48	Exchange email, SQL Server for SAP, fileserver, VMware
1,414,801,905	28	Exchange email, SQL Server for SAP, fileserver, VMware
368,800,504	24	Microsoft Windows 2008 R2 environment, mostly Hyper-V cluster
111,863,736	24	VMware and a handful of physical direct attached Wintel machine
2,673,873,236	24	ESX datastores volumes, SQL servers and, test Oracle databases
1,735,334,707	15	Billing Application
1,807,470,033	48	SAP on IBM system-i (AS400)
1,000,000,000	24	Bank
2,262,834,499	48	SAP R3 database
452,256,690	16	Oracle database, financial application used to manage pension fun
286,133,498	24	VMware environment, SQL servers, and SQL/SAP
793,597,183	12	Application environment is EPIC (Health care)
1,934,444,757	24	EPIC benchmark
791,390,915	24	Database
818,677,441	24	Multiple AIX systems and some Win2k8 R2 SQL servers
538,795,437	16	Hosts AIX and VMware and Oracle database
1,407,151,765	24	Oracle/ZFS
2,279,587,775	52	VMWare ESX
634,121,863	4	SPC1 test
241,452,177	24	ESX vmware and GPFS file system used for analytics
497,588,635	24	VMWare environment
1,623,641,554	24	Windows, AIX, Oracle, SQL, and VMS
407,921,312	24	VMWare
574,777,286	24	SAP R3 database
211,647,690	24	Unknown
2,274,760,000	24	
1,083,752,234	24	
2,067,512,157	24	
1,023,943,436	24	
188,536,064	24	
1,023,616,635	19	
762,257,464	24	
331,531,816	24	14

Table 1: Representative trace list. In some cases, we do not know the applications accessing the storage.

The *capture ratio* is the fraction of IOs that land on a region that is in the sample. Figure 2 shows the distribution of the capture ratio vs. sample ratio. If the ratios are equal then IOs are uniformly distributed. In our traces, the average sample ratio is 10%, and the average capture ratio is 30%. In extreme cases, 50% of the IOs are concentrated on 10% of the accessed regions.

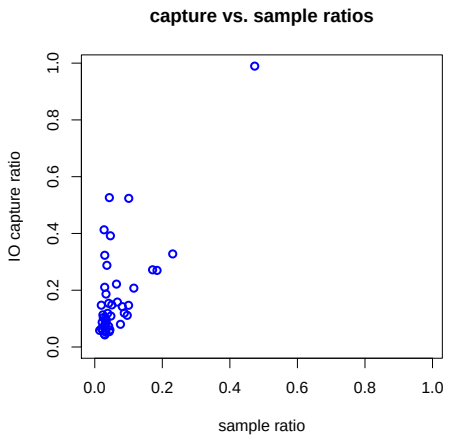


Figure 2: Distribution of capture ratio vs. sample ratio.

4.5 Performance

Figure 3 depicts simulation performance. The x-axis is simulation speed measured in thousands of IO per second, the y-axis is number of traces in each discrete bucket. The maximal speed is around 800,000 IOps, the minimal is a bit above 50,000 IOps, and the median is around 400,000 IOps. The XIV IOps in the field rarely goes above 40,000 IOps, because some IOs reach the disk, slowing applications down. The time to simulate a customer 24 hour trace is normally about an hour, even for heavy traces. Trace length is not the primary factor in simulation speed, rather, it is trace locality of reference. Our region data structures are allocated in 2GB of RAM, significantly larger than the processor L2 cache. If a burst of IOs touch a region, it will be resident in the L2. If IOs are spread across many regions, they will compete for L2 cache space, slowing everyone down.

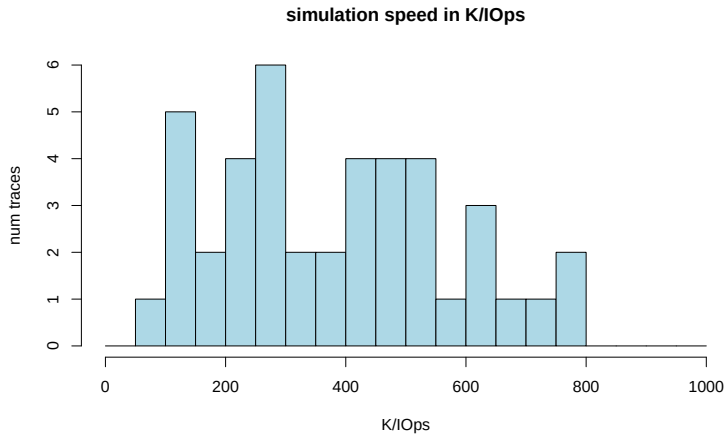


Figure 3: Histogram of simulation speed measured in K/IOps. The maximal speed we get is 800,000 IOps, the minimal is 50,000 IOps, and the median is around 400,000 IOps. It is rare for an XIV rack in the field to perform more than 40,000 IOps. This is because some IOs inevitably miss the cache, and have to be serviced by the disks.

The computational complexity is $O(|trace\ length| + K|num\ minutes|)$. Where K is a constant determined by the time it takes to perform the per-minute region scan. The simulation does not get slower as cache size increases, because the heap size stays constant. The factor that matters most is the number of live 16MB regions. In addition, accuracy is lost, as the sampling rate climbs. The read-write ratio does not affect the simulation significantly, because processing reads and writes costs roughly the same in terms of CPU cycles. The algorithm is just as accurate during warm up vs. steady-state.

4.6 Latency modeling

A customer's goal is generally not the hit rate as such, but the speed of the IO response, as reflected either in the average latency of the operations (which is crucial to response time for online applications) or in the IO rate that can be achieved (which affects the windows needed for batch job completions). The two are closely related; most batch processes, the IO rate increases significantly when IO latencies go down. It is therefore very useful to deliver estimates of IO latencies with and without SSD caching.

Our simple approximation is to estimate a new average latency for each measurement interval, by using the predicted hit ratio to interpolate between the observed average latency at that time for misses (i.e., disk reads) and a fixed latency value for SSD hits. This estimate is not precise: It ignores the reduction of disk queues as disk reads are reduced, and any changes in CPU operations and internal network transfers. Even if latency of the storage system were modeled precisely, it would nevertheless not be a precise what-if result because we do not know how much the arrival rate and timing of IOs will change as applications respond to shorter average latencies.

4.7 Laboratory validation

We conducted a number of replay experiments to validate model results against the behavior of the actual product caches against an identical IO stream.

Figure 4 presents original, prediction, and replay results for a database workload. The trace covered a 24 hour time frame on a customer storage rack that serves as the backend for a production database used by an SAP installation. The rack was a Gen2 machine with 15 modules and 120GB of RAM cache. The original hit rate was around 60%. The replay and prediction timelines closely match, during warmup and steady state. They reach around 90% after a day's warmup.

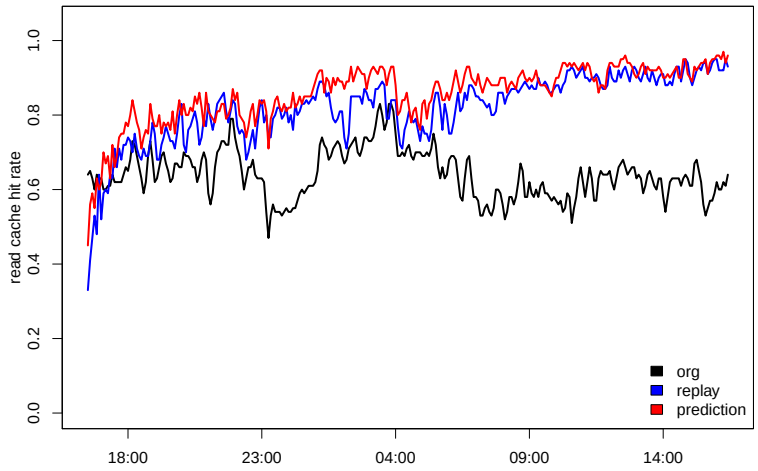


Figure 4: Replay experiment with a database workload. The black line shows the original read cache hit rate, and the red line shows the prediction. The blue line shows actual results when the trace was replayed against a Gen3 machine with an SSD. The original customer machine was a Gen2, and we can see that moving to a Gen3 with SSD is expected to increase hit-rates from 60% to 90% after a day's warmup.

Figure 5 presents results for an additional database workload. The trace covered a six-hour time, on a Gen2 rack with 14 modules and 112GB of RAM cache. The original hit-rate (without SSDs) is around 30%. The overall prediction for a system with SSDs, in red, ends up at the 80% mark after a four hour warm up period. A replay of the workload on a Gen2 system in the lab with SSDs, in blue, was very close to the prediction.

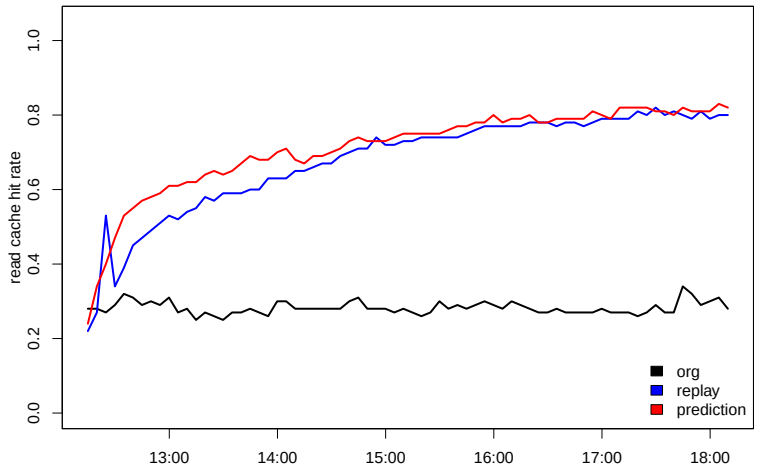


Figure 5: Replay experiment with a database workload. Hit rates improve from 30% to 80% during a six hour period.

The remarkable thing about this trace, is that it has three regions that account for 12% of the IOs. Ignoring these hot spots lead to 10% hit-rate prediction errors.

5 Field experience

The following is a case study of a customer that initially had a Gen2 machine, replaced it with a Gen3, and finally added SSDs. The application was an Oracle^{TM} database. Figure 6 shows the progression in read cache hit rates. The top graph show the Gen2 rack, the middle shows the Gen3, and the bottom depicts Gen3 with SSDs.

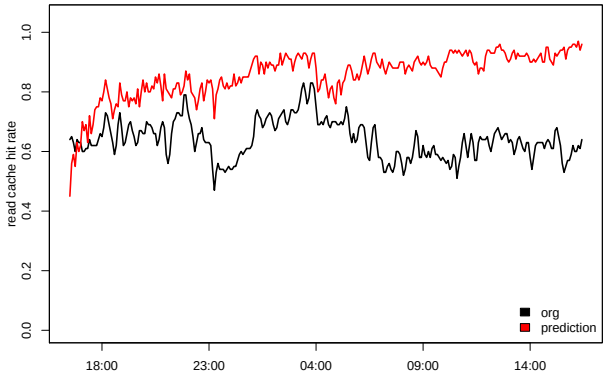
Initially, the customer had a Gen2 machine with an average IOps of 7000, and a read cache hit rate around 50%. The prediction was that, with the addition of SSD, a 90% cache hit rate would be achieved (this was an early version of the cache prediction algorithm). The projected improvement was sufficient to warrant the effort of moving the workload onto a system with SSDs. After moving to Gen3, performance hadn't changed much, and cache read hit rates were still around 50%. The updated SSD prediction was for a 95% cache hit rate. The customer decided to purchase SSDs, this increased the average IOps to 13000, and achieved the 95% read cache hit rates within two days of installation. The bulk of the workload was random 8KB reads

to a 3TB disk area. Since this fit into the SSD, performance dramatically improved.

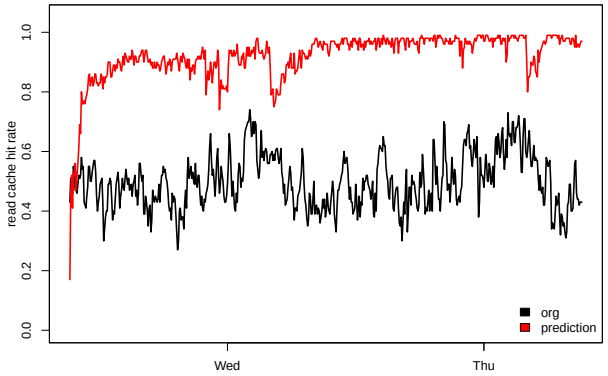
It is instructive to compare, in Figure 7, the measured hit rates before and after SSDs were installed. Note that the workload with SSDs was significantly heavier. The predicted hit rate (solid blue) has a steady warm-up for about 6 hours in the first day (a), after which the predicted rate is high (about 92% near 6:00 PM at (b)) but visibly lower than the rate in the real system at the same time of day (97%). However, the predicted hit rate in the second day is markedly higher, and closer to the observed rates. This does not indicate the second day differs from the first, but rather that content referenced in the second day is left in cache from the first day, so the warmup was not fully complete when the hit rate first seemed to level off.

In comparing prediction and measurement for the second day, there are discrepancies labeled (d) and (e) which are associated with overnight batch processes, whose size and timing may be different in traces taken months apart. There are also many times where the two are remarkably close (c), but the apparent precision might arise from compensating errors. In Figure 6(Gen3 + SSD), where the same measured hit rate is shown with the predicted hit rate *from the same trace*, we actually see a greater difference at the times corresponding to (c), typically of several percent. This is well within our target for accuracy, and is typical of the achieved accuracy.

Gen2



Gen3



Gen3 + SSD

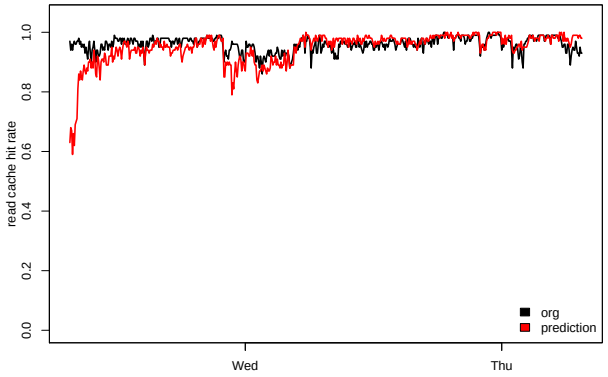


Figure 6: A customer that migrated from Gen2, to Gen3, and then Gen3 with SSDs. The prediction was for a large improvement in read cache hit rates, and read latency. The graphs show the progression in cache hit rates, from Gen2 at the top, to Gen3+SSD at the bottom.

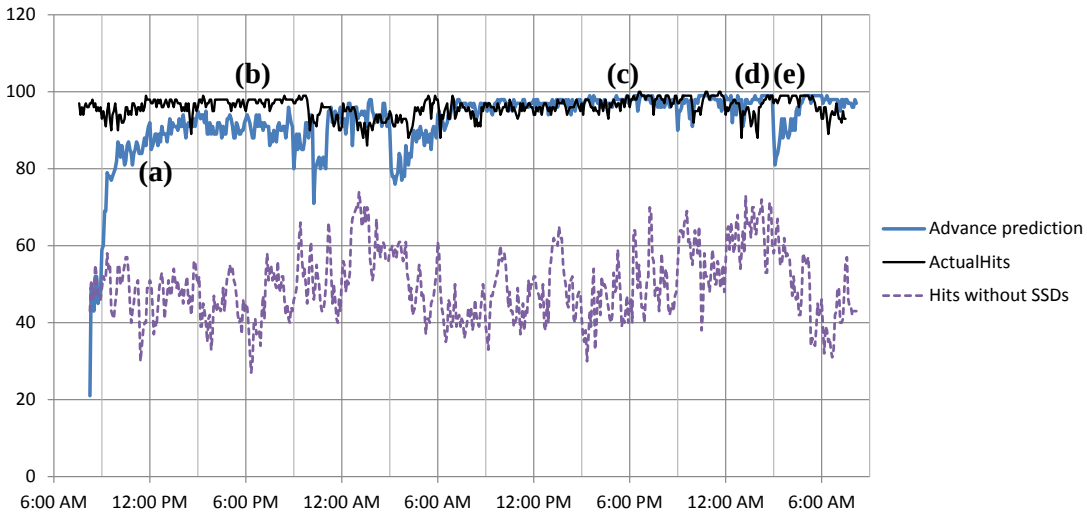


Figure 7: Comparison of advance prediction with customer results.

The significance of day-to-day access repetitions is verified in the time distribution we measured for consecutive accesses of the same page (Figure 8). About 455 GiB of data is read during the first day and not accessed again until it is read 24 ± 0.5 hours later. Another conclusion from the curve in Figure 8 and others like it is that very high accuracy in cache modeling (better than 1%, say) depends on handling the tail of the re-access distribution, and thus on modeling well the retention of cache content across many hours or days. For our users, an error of a few percent does not matter in the decision to deploy an SSD cache. However, for the related problem of designing a cache policy, retaining all the right multi-day content might change 95% hits to consistently more than 99%, and the $5\times$ decrease in miss rate would dramatically increase performance.

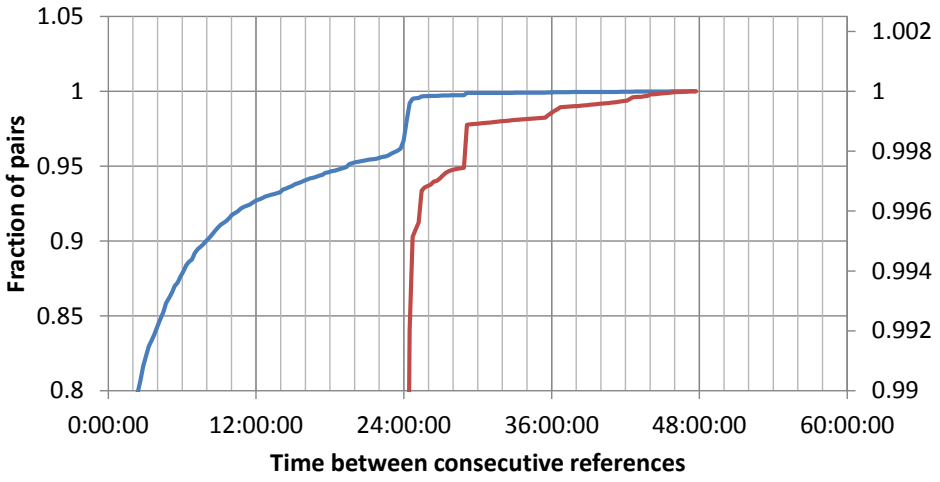


Figure 8: Cumulative distribution of time differences between consecutive reads of the same page. The distribution applies to the workload used for the prediction in Figure 7. Only reads without intervening writes are counted. More than 20% of observed re-accesses have spacings greater than 2 hours. The blue-curve step at 24:00:00 is 455 GiB read and re-read 24 hours later. Red curve has enlarged scale to show additional, much smaller, re-reference features near spacings of 25:20, 29:00, and 36:00 (hh:mm).

Table 5 presents the accuracy of the predictions, in a number of cases where they could be verified. These are customer machines, running production workloads, that have SSDs. This allows comparing the simulation predicted cache hit rate, with the actual hit rate. The duration of the workloads varies between 11 hours, and a full week. The columns represent the following information:

duration	duration of trace hh:mm:ss
caHit	read cache hit percentage, after warmup
simCaHit	predicted read cache hit rate, after warmup
MsqrErr	mean squared error
avgReadLtnMS	average read latency in milliseconds
seq	percent of sequential IO
read	percent of read IO
iops	average IOs per second
description	customer applications and setup

The caHit and simCaHit measurements refer to the second half of the

workload, after the simulation has warmed up. The *caHit* column shows the median of the actual cache hits. The *simCaHit* column presents the median of the simulated cache hits. The error is mostly lower than 5%, although we have some outliers. The average read latency, IOps, and other workload characteristics vary between workloads.

duration	%caHit	%simCaHit	MsqrErr	avgReadLtnMS	%seq	%read	iops	description
12:09:59	70	66	4.53	4.31	13	80	18256	Healthcare
24:00:00	39	42	11.63	10.77	22	72	3276	VMWare, SQL servers supporting SAP
15:00:02	95	95	1.51	2	39	77	32243	Billing application
16:11:18	76	75	6.34	6.1	28	55	9224	AIX, VMware, Oracle database
24:00:03	82	83	4.17	3.79	30	55	3736	
48:00:02	96	97	2.14	0.93	40	84	13057	SAP R3 database
24:00:02	80	79	14.97	3.33	56	79	10499	DB2, several SQL servers and VMware clie
52:44:00	94	94	3.62	0.93	25	22	11968	VMWare ESX
11:39:23	96	94	3.98	0.66	10	66	7451	VMWare ESX
169:46:00	97	97	2.28	0.45	37	20	4642	VMWare ESX
24:09:49	97	96	2.24	0.4	17	49	5622	
24:10:13	95	97	3.58	0.85	12	64	2691	VMWare ESX, and analytics

Table 2: Result summary for cache prediction, over a group of customer machines.

There are cases where the prediction is not very good. This can happen if the RAM cache hit rate is dominant. For example, Figure 9 shows Exchange mail workloads. They have a highly sequential component, which is handled entirely by RAM, bypassing the SSD. The random IO component mostly fits in RAM. Taken together, this means that RAM is the dominant cache, and SSD accounts for very little cache hits. Such cases are not very common, however, they can cause high prediction inaccuracy. In this case, the prediction *with* SSD underestimates the hit-rate *without* SSD by around 10%.

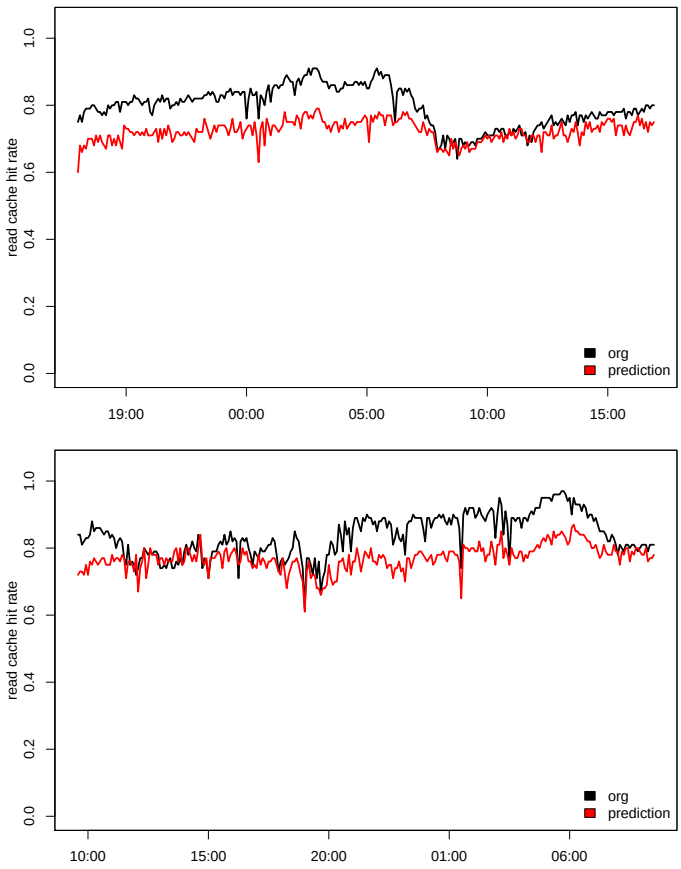


Figure 9: Exchange workloads. The prediction is lower than the current hit rate. This is because the dominant factor in caching is RAM, and the RAM simulation is much less accurate than the SSD simulation.

6 Summary

This paper described a study in IO tracing and analysis. The study was conducted on a large body of block traces collected from IBM XIV enterprise storage systems in production use. Our cache prediction method is simulation based, and has successfully been used by dozens of customers in the field. It can, on average, predict cache hit rates with 5-10% accuracy.

We contribute methods to:

1. Estimate sequentiality as well as page re-reference.

2. Sample the IO stream, and maintain memory overhead flat. This uses the idea of *address-space* sampling in regions, and could come at the cost of reduced accuracy.

We found that real workloads could be difficult to sample, because access can be highly skewed. We believe these methods, and our statistical approach, could prove beneficial for other systems and caching algorithms.

7 Acknowledgments

We would like to thank Theodore Gregg, Anthony Vattathil, and Aviad Offer for their help and support.

References

[1] B. Anderson. Mass storage system performance prediction using a trace-driven simulator. In *22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 297 – 306, 2005.

[2] B. Dufrasne, I. K. Park, F. Perillo, H. Sautter, S. Solewin, and A. Vattathil. *Solid-State Drive Caching in the IBM XIV Storage System.* International Business Machines Corporation, USA, 2012.

[3] C. M. Thomas, M. A. Hirsch, and W. M. W. Hwu. Combining Trace Sampling with Single Pass Methods for Efficient Cache Simulation. *IEEE Transactions on Computers*, 47:714 – 720, June 1998.

[4] C.A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Operating Systems Review*, 36(SI):181–194, December 2002.

[5] D.K. Tam, R. Azimi, L.B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. *SIGPLAN*, 44(3):121–132, March 2009.

[6] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami. Cost Effective Storage Using Extent Based Dynamic Tiering. In *9th USENIX Conference on File and Stroage Technologies*, FAST'11, Berkeley, CA, USA, 2011. USENIX Association.

[7] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. *SIGMETRICS Perform. Eval. Rev.*, 18(1):134–142, April 1990.

[8] J. Wolf, H.S. Stone, and D. Thiébaut. Synthetic Traces for Trace-Driven Simulation of Cache Memories. *IEEE Transactions*, 41(4):388–410, April 1992.

[9] M. D. Hill. Aspects of Cache Memory and Instruction. Technical report, Berkeley, CA, USA, 1987.

[10] P. Updike and C. Wilson. TR 3832, Network Appliance, Sunnyvale, CA, USA, September 2011.

[11] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. *SIGOPS Operating Systems Review*, 38(5):177–188, Oct 2004.

[12] P.J. Denning. The Working Set Model for Program Behavior. *Communications ACM*, 11(5):323–333, May 1968.

[13] Y.H. Kim, M.D. Hill, and D.A. Wood. Implementing Stack Simulation for Highly-associative Memories. In *ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 212–213, New York, NY, USA, 1991. ACM.