Warming up Storage-Level Caches with Bonfire

Yiying Zhang†, Gokul Soundararajan*, Mark W. Storer*, Lakshmi N. Bairavasundaram*,
Sethuraman Subbiah*, Andrea C. Arpaci-Dusseau†, Remzi H. Arpaci-Dusseau†

† *University of Wisconsin-Madison,* * *NetApp, Inc.*

Abstract

Large caches in storage servers have become essential for meeting service levels required by applications. These caches need to be warmed with data often today due to various scenarios including dynamic creation of cache space and server restarts that clear cache contents. When large storage caches are warmed at the rate of application I/O, warmup can take hours or even days, thus affecting both application performance and server load over a long period of time.

We have created *Bonfire*, a mechanism for accelerating cache warmup. Bonfire monitors storage server workloads, logs important warmup data, and efficiently preloads storage-level caches with warmup data. Bonfire is based on our detailed analysis of block-level data-center traces that provides insights into heuristics for warmup as well as the potential for efficient mechanisms. We show through both simulation and trace replay that Bonfire reduces both warmup time and backend server load significantly, compared to a cache that is warmed up on demand.

1 Introduction

Caches are useful only when they contain data. Cache warmup, the process of filling caches with data, has so far received scant research attention since it occurred rarely and did not have a significant impact. Thus, cache warmup simply happens on-demand, that is, as the workload performs I/Os. Today, the frequency and the impact of cache warmup are both changing.

Cache warmup of server-side, or storage-level, caches now occurs more often. For example, cache space is being created dynamically in order to improve application performance [5] or reduce backend I/O load [21]. These caches take time to become effective due to warmup [5]. As another example, with increasing scale, many planned and unplanned restarts of storage nodes occur [14]; their caches need to be warmed up after the restart.

Cache warmup also has a tremendous impact on service-level agreements (SLAs) since storage-level cache space is now extremely large – typically hundreds of Gigabytes of DRAM and Terabytes of flash [12, 22, 23]. SLAs for performance and availability often use wall-clock time to define degraded mode [4]. Terabyte-sized caches need a large number of I/Os to become warm. When these I/Os are issued on-demand at the rate of application I/O, the rate may not be high enough to fill the cache quickly (with respect to wall-clock time).

Given the increase in both the frequency and impact of cache warmup, we need to develop *effective* and *efficient* warmup mechanisms. An effective mechanism will warm the cache with data that will be used by the application, thereby increasing the cache hit rate as compared to a cold cache being warmed through application access. The increased hit rate will improve the average I/O latency experienced by the application, as well as reduce I/O load on backend storage. An efficient mechanism will use minimal overhead when monitoring workloads for warmup-related characteristics and also place less load on the storage backend during warmup process (called *pre-load*) than on-demand warmup.

We develop a mechanism called Bonfire to warm storage-level caches effectively and efficiently. Bonfire monitors storage server workloads, logs valuable warmup information, and bulk loads warmup data to new caches. In creating Bonfire, we make three contributions.

First, we analyze a variety of traces and quantify workload characteristics that are important for cache warmup. We find two primary patterns of block reaccesses: reaccesses within an hour and daily reaccesses that happen at the same wall-clock time each day. We also find correlations between spatial and temporal locality.

Second, based on our trace analysis, we design a set of cache-warmup algorithms and show their benefit through simulation. We also construct a decision tree of cache warmup heuristics based on the simulation results. We show that using the most-recently accessed data for warmup provides the best general-case warmup effectiveness. For episodic patterns, such as running a series of regression tests, using the data from the same time period in the previous day provides the most effective cache warmup. If the cache workload is well understood, the decision tree can be used to determine the optimal cache warmup approach.

Third, we implement and evaluate a prototype of Bonfire that uses the most-recently accessed data for warmup. Bonfire has two options for recording information and performing warmup – *metadata-only* and *metadata+data*. *Metadata-only* logs only the LBAs (block addresses) that are accessed, while *metadata+data* also logs the accessed data sequentially on a separate disk or SSD. In the former case, Bonfire obtains the data for warmup from the backend disk location. In the latter case, the data is read from the logging device. The difference between the two options is in the time taken for pre-load and in the overhead for logging. We evaluate both options in our experiments.

With Bonfire, after warmup, we see an overall speedup of up to 100% in warmup time (i.e., time until cache performance matches that of an "always-warm" cache) and up to a 228% reduction in server load as compared to on-demand warmup. Meanwhile, Bonfire adds only small overhead to the storage system. The *metadata-only* option uses 256 KB of memory and up to 71 MB of logging device space. It also takes 3 to 20 minutes to pre-load warmup data. The *metadata+data* option uses 128 MB of memory and up to 36 GB of logging device space. It takes 2 to 11 minutes to pre-load warmup data.

2 Storage-level Cache Warmup

Caches have been used in storage systems for a long time; research in storage caching has also addressed a variety of problems including replacement algorithms [8, 16, 28], prefetching algorithms [24], cache partitioning [27], etc. However, one area that has received little attention is the issue of cache warmup, that is, the process of filling the cache with data useful to the workload. The typical approach for warming caches is to allow workloads to warm the cache on-demand, as part of normal I/O.

Three important developments cause us to re-examine cache warmup. First, storage-level caches today are extremely large; enterprise storage systems contain hundreds of Gigabytes of DRAM and Terabytes of flash, used primarily for caching [12, 22]. Second, non-disruptive operation is now expected in many data-center environments. Third, given the size of today's datasets, operating with a cold cache would cause applications to violate their service level objectives (such as, average latency); large violations may be considered akin to unavailability. Maintaining warm caches is essential for handling the confluence of the aforementioned trends.

2.1 Cache Warmup Scenarios

In the modern storage server environment, storage-level cache warmup is not uncommon. We list a few scenarios where storage-level caches can be cold.

Dynamic caching: In a clustered storage system, I/O requests are handled by a distributed set of storage servers, each containing a fraction of the overall data [6, 15]. Regardless of the data distribution scheme, hot spots can occur [18]. A common solution to handle hot spots is to deploy a cache to offload the I/Os from them [7, 21]. Specifically, cache space is created on a node to process I/O requests for a hot node. Such newly created cache space is in a cold state. In order to meet service levels and offload I/Os, the cache needs to be warmed up quickly.

Server restart: Scheduled and unscheduled restarts of storage servers do occur and they need to be handled quickly and efficiently. After the restart, the system needs to return quickly to the performance level it was at prior to the restart. The storage-level cache space needs to be brought back to the warm state before the level of performance is back to normal. Although persistent and durable caching solutions do exist for flash-based caches, they have their own overheads and do not work for DRAM-based caches [26].

Server take-over: To handle server down time better, a common technique used is pairing storage controllers with a shared set of disks in the backend, or *high-availability pairs*. In the event of a controller failure, the operational storage controller takes over the duties of the failed controller. This approach is also useful for providing non-disruptive controller upgrades and maintenance. Storage-level cache warmup is an important issue in the high-availability scenario as well. The storage-level cache located in the operational storage controller needs to be warmed up with the content of the failed controller.

2.2 Impact of Storage-level Cache Warmup

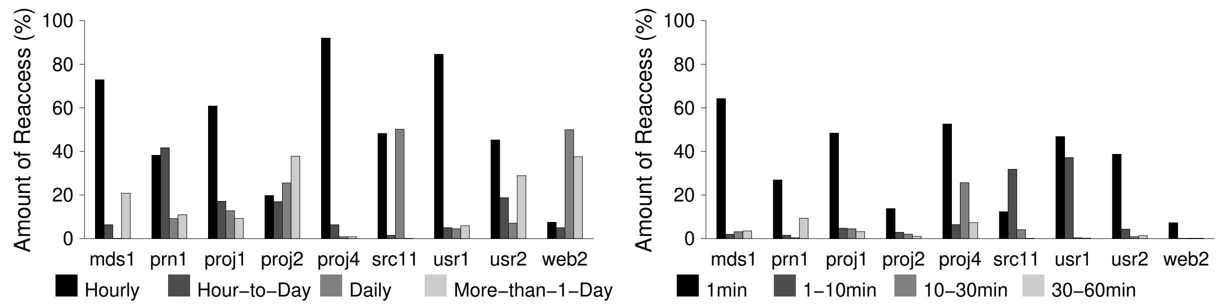
Storage-level caches are important both for improving application performance and for reducing server load.

Impact on application performance: When a storage-level cache is cold, application requests are served from slower storage (e.g., hard disks). Since there are usually a few orders of magnitude difference between DRAM- or flash-based caches and slower backend storage, applications will experience a significant performance drop in both latency and bandwidth, resulting in service level agreement violations. As we will see from our trace analysis in Section 3, the warmup time for typical data-center workloads can be hours to days. During this period, application performance will be worse than what a system with a storage-level cache can deliver.

Impact on server load: Another type of impact of the storage-level cache warmup period is on the I/O load going to the backend storage (either slower storage or a hot-spot). When the cache is cold, requests are served from the backend; when the cache is warm, this load is significantly lower. While pre-loading warmup data to the cache involves I/O load to the backend, the advantage is that such load is known in advance; the I/Os could be performed more efficiently or scheduled for the right time.

2.3 Our Approach

Our goal in designing the Bonfire system is to target most of the general storage-level cache warmup scenarios instead of a solution for a single scenario. To this end, we examine a variety of traces on key characteristics that impact the design of warmup mechanisms. We design Bonfire based on the analysis of traces. Using Bonfire, we aim to reduce the cache warmup impact on both application performance and server load.



(a) Hour Granularity

(b) Within an Hour

Figure 1: **Relative Reacess Temporal Pattern**

Name	Function	WSS (GB)	Size (GB)	Reaccess (%)
mds1	media server	84	95	10
prn1	print server	80	212	51
proj1	project dir. 1	642	870	16
proj2	project dir. 2	390	1298	55
proj4	project dir. 4	122	170	26
src11	source control	119	1521	90
usr1	user home dir. 1	608	2311	69
usr2	user home dir. 2	362	483	16
web2	web server	67	283	76

Table 1: **Selected MSR-Cambridge Traces** *WSS denotes the total working set size of the trace, which is the size of all the unique data read or written over the trace. Size represents the total amount of data in the trace. Reaccess percentage represents the fraction of total amount of data that is reaccessed.*

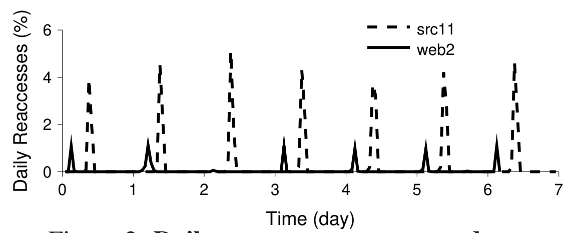


Figure 2: **Daily reaccesses over seven days**

3 Trace Analysis

We study traces from server environments to identify features that enable efficient bulk warmup of large storage-level caches. We want to understand the behavior of reaccesses (read-after-reads and reads-after-writes) along two dimensions: *temporal behavior* and *spatial behavior*.

3.1 Description of Traces

We use block-level traces released by Microsoft Research Cambridge [19]. These traces were collected from the MSR-Cambridge data center servers and are block-level one-week long traces starting from 5PM on February 22nd 2007. They were collected below the server buffer caches and there were no storage-level caches in the system [20]. We selected these traces for a number of reasons. First, they cover a variety of workloads exhibiting diverse access patterns. These workloads are server-level workloads, each containing multiple concurrent clients. Second, the traces cover a week of continuous usage; long trace times allow us to experiment with long warmup approaches and study long-term temporal behavior. Third, they are publicly available, fostering reproducible results and independent verification.

While the MSR-Cambridge traces serve our purpose of studying workload behavior for warming up storage-level caches that are added below the server buffer caches, they are not fit for other scenarios, e.g., client-side cache warmup or application-based cache warmup; these scenarios can be studied with traces that are collected above server buffer caches.

There are a total of 36 traces (13 servers, 36 volumes) in the MSR-Cambridge trace set, of which we select nine for close examination (6 servers, 9 volumes). We winnow the full set down to nine by filtering out those traces that are ill-suited for large caches; we first remove 16 write-intensive traces (>50% writes), then remove 10 traces that have a small working set size (WSS $<$ 32GB), and finally remove one trace with low reaccess rate ($<$ 1%). Table 1 summarizes the traces of our study. These traces include a variety of types, working set sizes, total I/O sizes, and reaccessed amount.

3.2 Temporal Reaccess Behavior

We begin our trace study by asking the following questions related to temporal reaccess behavior.

1. *What is the time difference between a reaccess and the previous access to the same block?*
2. *When do reaccesses happen (wall clock time)?*

We quantify temporal reaccess behavior by measuring the time between a reaccess and its previous access to the same block. We find two common patterns across all traces: reaccess within one hour of the previous access (*hourly*) and reaccess around one day (23 to 25 hours) after the previous access (*daily*). We plot the percentage of reaccessed logical block addresses that fall into these two patterns and the rest of them (*other*: from one hour to one day and more than one day) in Figure 1(a). Note that a logical block can belong to more than one pattern. For example, a logical block address can be accessed everyday at midnight and at 12:30 am; it exhibits both daily

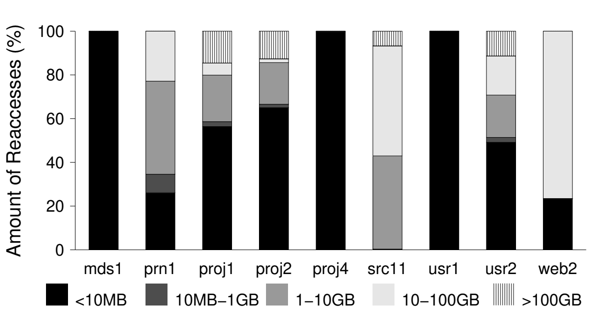


Figure 3: **Reuse Distance**

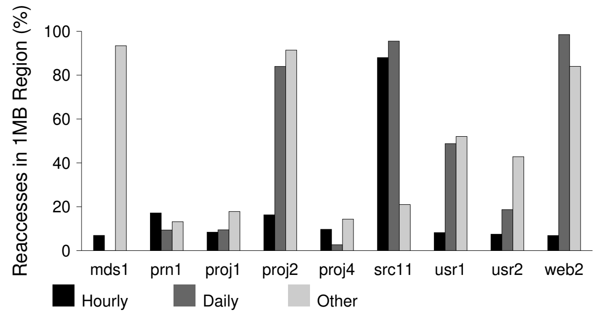


Figure 4: **Consecutive Length**

and hourly patterns. We find that the *mds1*, *proj1*, *proj4*, *usr1*, and *usr2* traces contain mostly hourly reaccesses. The *src11* and *web2* traces exhibit a daily access pattern. The *prn1* and *proj2* traces contain more reaccesses that do not fall into the either the hourly or the daily patterns.

The hourly pattern suggests that much of the data accessed in the last hour is very likely to be accessed next and that we can use the data accessed within the last hour to warm up a new cache. Furthermore, within the last hour, we find that more recently accessed data is more likely to be reaccessed next, suggesting that an LRU-like filtering would work well (see Figure 1 (b) for reaccesses within the last hour).

The daily pattern suggests that we can use the data accessed 24 hours ago to warmup the new cache now. We also study the relationship between daily reaccessed workloads and wall clock time in Figure 2. We find that daily reaccesses happen at the same time each day. Such regular patterns suggest batch or script workloads; based on the description of *src11*, we suspect that nightly regression tests are run on the source code repository.

3.3 Spatial Reaccess Behavior

We now study the spatial reaccess behavior and ask the following questions. The answers to these questions suggest whether we can use logical addresses to determine what data is valuable to cache warmup. Understanding Questions 3 and 4 also helps with the design of Bonfire data structures.

3. *How much distinct data is accessed between a reaccess and the previous access to the same block?*
4. *Is there any spatial clustering among reaccesses?*
5. *Where are reaccesses in the logical address space?*

We begin our study of spatial reaccess behavior by measuring the *reuse distance* of reaccesses. The reuse distance is measured as the number of distinct data blocks read or written between an access and its subsequent reaccess to the same block; shorter reuse distances imply that a small LRU-like cache would capture most reaccesses. We categorize the reuse distance into four groups and plot the percentage of reaccesses that fall into each group for each trace in Figure 3. It shows that many of the traces have very small reuse distances; six of the nine traces have a majority of reaccesses within 10 MB. This result correlates well with Figure 1a; as expected, we find that hourly reaccesses have shorter reuse distances and daily reaccesses have longer reuse distances. Overall, the results suggest that the amount of data needs to be monitored for the hourly pattern is small.

Next, we study the spatial clustering of reaccesses. Spatially clustered reaccesses, i.e., a contiguous segment of blocks reaccessed together, imply coarser logging granularity and faster warmup data pre-loading from backend data disks. To measure spatial clustering for each reaccess category (hourly, daily, and other), we look at all the 1 MB regions that contain reaccesses of that category and calculate how many 4 KB blocks in the 1 MB region are reaccesses of that category. We then take the average of all such percentages across the 1 MB regions and plot them in Figure 4. By correlating with Figure 1a, we find that traces with a majority of hourly reaccesses tend to access blocks randomly while traces with predominantly daily reaccesses (i.e., *src11* and *web2*) access larger disk regions. Therefore, the results imply that hourly reaccesses should be tracked at a fine granularity while daily reaccesses can be tracked at a coarser granularity.

Finally, we find that reaccessed data spread over all the address space and there is no specific pattern in the region that is heavily reaccessed, i.e., no hot spots in the logical address space.

3.4 Key Observations

1. We find that reaccesses show two common temporal patterns: blocks tend to be reaccessed within one hour of the last access (*hourly*) and approximately a day after the last access (*daily*). Seven out of nine traces contain significant amount of hourly reaccesses. **Implication:** Keeping track of recently accessed data will benefit cache warmup.

2. Some workloads exhibit a clear daily reaccess pattern. Furthermore, the reaccesses happen at the same wall-clock time each day. **Implication:** Keeping track of data accessed within the last day may benefit cache warmup.

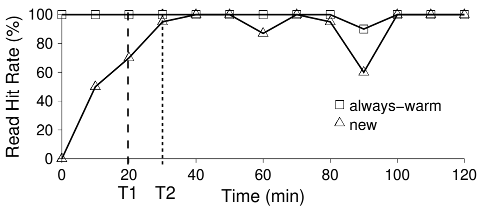


Figure 5: **Example for Bonfire Metrics**

3. We find that hourly reaccesses have small reuse distances. In some cases, non-hourly workloads also have small reuse distances. **Implication:** Even a small buffer of recently accessed blocks may provide a benefit for cache warmup.

4. We observe that daily reaccesses exhibit spatial clustering. **Implication:** Longer-term reaccesses can be tracked at a coarser granularity.

4 Cache Warmup Algorithms

Using the observations made in Section 3, we design a set of algorithms and perform experiments with them using a cache simulator with different configurations. The goal of our study is to find if bulk cache warmup is useful and what warmup data is important for different workloads. In this section, we describe the cache warmup metrics, the cache warmup algorithms, and the simulation results.

4.1 Cache Warmup Metrics

The goal of cache warmup is to both reduce the application perceived performance drop and reduce the I/O load on storage servers. We define two metrics that measure the performance with respect to our goal: *cache convergence time* and *server load reduction*. Cache convergence time is defined as the time from the new storage-level cache is added until the cache is fully warm, which can be viewed as the cache warmup period. Server load reduction measures the reduction of I/Os going to the storage server due to I/Os being handled by the cache during the cache convergence time.

We first define the cache convergence time. Our goal of cache warmup is to bring the cache into a state as if it has always been running. We compare the new cache behavior to an *always-warm* cache, a cache that has been running for long and thus in a warm state. We assume both caches use the same replacement policy, i.e., LRU in this study. Specifically, we compare the read hit rate of a new cache to an always-warm cache to see when they converge. The convergence time is the earliest instance when the new cache matches the performance of an always-warm cache (and continue to match till the end of the trace). As workloads vary over time and may not have fixed service levels, we loosen the definition to accommodate minor fluctuations using two parameters: *violation threshold* (Th_V) and *convergence threshold* (Th_N).

$$\begin{aligned} N\_violates^T &= \text{number of } t\ s.t. \\ & \quad T \le t \le T_{end} \text{ and } H^t_{new} < Th_V \cdot H^t_{warm} \\ T_{converge} &= \min\{T \mid N\_violates^T \le Th_N \cdot (T_{end} - T)\} \end{aligned}$$
$$\text{where } 0 < Th_V \le 1, 0 \le Th_N < 1 \text{ and } H^t_{scheme} \text{ is the read hit rate of a cache scheme at time } t \tag{1}$$

We define the convergence time, $T_{converge}$, in equation 1. In this equation, Th_V is the threshold that defines what is a violation of convergence and Th_N measures how many such violations ($N\_violates^T$) are allowed. We look at the time period after T till the end of the trace (T_{end}); the total amount of time when the new cache hit-rate is lower than Th_V of the always-warm cache hit-rate needs to be smaller than Th_N of this period. We define the smallest of such T as $T_{converge}$. $T_{converge}$ measures how fast a cache can go into a warm state according to wall clock time. Reducing $T_{converge}$ results in meeting user expected application behavior or SLAs faster.

Figure 5 gives an example of how we calculate $T_{converge}$. We show a sample graph of read hit-rates over time for an always-warm cache and a new cache. There are 13 time intervals from the start of a cache till the end of the trace. $T1$ shows the convergence time when the threshold Th_V is set to 90% and Th_N is 20%. The number of hit-rate violations between $T1$ to T_{end} is 2, which is smaller than 20% of the points in the period from $T1$ to T_{end}. $T2$ is the convergence time when Th_V is 80% and Th_N is 10%; in this period, there is 1 violation at 90 min.

$$R_{server\_load} = \frac{\sum_{t=0}^{T_{converge}} (Data^t_{cache})}{\sum_{t=0}^{T_{converge}} (Data^t_{cache} + Data^t_{server})} \tag{2}$$

Our second metric is the reduction of server I/O load during the warmup period ($T_{converge}$). We define the reduction of server I/O load as the fraction of data that is served by the cache out of the total data issued during the warmup time as in Equation 2. To compare across different warmup schemes, we always use the same warmup period, the cold convergence time, over which the server reduction is calculated. R_{server_load} measures how effective a cache is in reducing server load during warmup. A higher value means most of the data is served from the cache and the server has light load. Apart from the goal of reducing cache warmup time, Bonfire also tries to improve the performance before a new cache is warmed. R_{server_load} serves as a metric to evaluate cache performance during such degraded mode.

4.2 Algorithms

In this subsection we describe the warmup algorithms we select, and how we select them. The results in Section 3 help us design several cache warmup algorithms: *Implication-1* suggests that keeping track of the recently

Parameter	Category	Value
Cache Size S_{Cache}	System Setting	25, 50, 75, 100%
Warmup Size S_{Warmup}	Bonfire Setting	5, 10, 25%
Warmup Region Size	Bonfire Setting	4 KB, 1 MB
Converge Th_V	Metrics Threshold	70, 80, 90%
Converge Th_N	Metrics Threshold	1, 3, 5%

Table 2: **Simulation Parameters** *Cache size and warmup size are percentage of working set size of each trace.*

accessed blocks may benefit warmup, *Implication-2* suggests that the I/O accesses from the previous day may benefit warmup.

Last-K: This scheme tracks the last k regions accessed in a trace. This algorithm is designed for hourly reaccesses that exhibit a high degree of reuse in the recent past; since Last-K chooses the most recently accessed regions, it imitates an LRU state thereby matching our goal to warmup the cache to be similar to an always-warm LRU cache.

First-K: This scheme tracks the first k regions of the past 24 hours. This algorithm is designed for daily reaccesses, since it chooses the same time period from the previous day to warm up the current day.

Top-K: This algorithm tracks k most frequent regions accessed in the past 24 hours. This algorithm is a frequency-based algorithm. It serves as a comparison algorithm when no temporal trace behavior knowledge is known.

Random-K: This scheme simply tracks k random regions accessed in the past 24 hours. This algorithm serves as a comparison algorithm when we have no prior knowledge of the trace behavior.

We compare the performance of each algorithm with a cold cache that is warmed up on demand and an always-warm cache. We also consider disk prefetching and compare all algorithms with and without disk prefetching

4.3 Simulation Results

We make three observations with our simulation study. First, cache warmup reduces the time needed for a new cache to match the behavior of an always-warm cache. Second, even during cache warmup, server load is reduced compared to a cold cache. Third, certain workload features can be used to match the workload to the most effective warmup algorithm.

4.3.1 Simulation Experiments

We replay traces using a cache simulator with LRU replacement policy. We split each week-long trace into seven day-long traces; all our algorithms use the data from the previous day to warmup the current day, since from our analysis in Section 3 most reaccesses happen within one day. We vary several parameters in our simulation: cache size, warmup data size, warmup region size, and convergence thresholds. The configuration parameters are listed in Table 2; we choose a small set of reasonable values for each parameter and perform experiments with each algorithm on all combinations of the parameters.

4.3.2 Overall Improvement

We first present the overall results of cache warmup and the effect of different configurations on cache warmup. Table 3 shows the results of the convergence time and the server load reduction with different configurations. Server load reduction is measured over the convergence time period of the cold cache for all schemes. We perform experiments with all combinations of parameter values but only present the smallest and the biggest values of each parameter in the table. For each configuration, we select the result of the best algorithm for each trace and then calculate the median values over all traces.

The pre-load time Bonfire uses to load warmup data into the new cache is dependent on real storage system and data layout. As an optimization for fast warmup data load, we propose the use of a performance snapshot in Section 5 to store the warmup data in a contiguous space and load it in bulk. For the simulation, we perform a simple estimation of Bonfire pre-load time ($T_{preload}$) by using 100 MB/s hard disk sequential read throughput and the warmup data size. With this calculation, the median pre-load time is 5.2 minutes when warmup data is 25% of the working set size (WSS) of the traces and 1 minute when warmup data is 5% of the WSS; in both cases, pre-load time is much smaller compared to the convergence time (hours). We explore more details about the actual warmup data pre-loading time with real experiments in Section 6.3.

Overall, we find that Bonfire reduces both convergence time and server I/O load over the on-demand approach (*cold*). Bonfire improves the convergence time by 14% to 100% (Column 8 in Table 3) and has 44% to 228% more server load reduction (Column 12 in Table 3) than cold.

Comparing different parameters, we find that Bonfire has bigger improvement over cold when the cache size is smaller. Specifically, using Th_V of 90% and Th_N of 5%, the average Bonfire convergence time improvement for a small cache is 90% and the improvement for a large cache is 65%. Bonfire also reduces the server load for small caches; using Th_V of 90% and Th_N of 5%, the average load reduction is 75% for a small cache and 63% for a large cache. As expected, bigger warmup size increases the pre-load time but results in better Bonfire warmup, since more data is brought into cache for warmup.

A lower violation threshold Th_V and a higher convergence threshold Th_N lead to a looser convergence time criteria, resulting in better Bonfire improvement in terms of convergence time. For example, the average convergence time improvement for Th_V of 90% and Th_N of 5% is 77% and the average improvement using Th_V of 90% and Th_N of 1% is 42%. We also find that 1 MB region size gives better results than 4 KB region size.

We also evaluate the impact of disk prefetching with all the schemes; disk prefetching uses readahead of a 128 KB

S_{Cache}	S_{Warmup}	Th_V	Th_N	$T_{preload}$ (min)	$T_{converge}$ Cold (hr)	Bonfire (hr)	Improve (%)	Server Load Reduction (%) Cold	Always-Warm	Bonfire	Improve
100	25	90	5	5.2	19	4.6	76	33	99	70	112
			1	5.2	21	17	19	34	99	59	79
		70	5	5.2	18	3.7	79	28	99	70	150
			1	5.2	21	11	48	33	99	63	91
	5	90	5	1.0	19	8.9	53	33	99	56	70
			1	1.0	21	18	14	34	99	49	44
		70	5	1.0	18	5.0	72	28	99	55	96
			1	1.0	21	18	14	33	99	49	48
25	25	90	5	5.2	18	0	100	27	88	83	207
			1	5.2	19	5.7	70	25	74	78	212
		70	5	5.2	8.4	0	100	24	88	77	221
			1	5.2	19	5.3	72	25	70	82	228
	5	90	5	1.0	18	3.6	80	27	88	67	148
			1	1.0	19	6.8	64	25	74	65	160
		70	5	1.0	8.4	0	100	25	88	64	156
			1	1.0	19	7.0	63	25	70	68	172

Table 3: **Overall Simulation Results** *Warmup region size is set to 1 MB for all results in the table. $T_{preload}$ denotes the time (in minutes) Bonfire uses to load the warmup data into the new cache. Bonfire $T_{converge}$ Improvement is calculated as the percentage of Cold $T_{converge}$ minus Bonfire $T_{converge}$ and Bonfire $T_{preload}$ over Cold $T_{converge}$. Server Load Reduction is calculated during the Cold $T_{converge}$ time for all schemes (Cold, Always-Warm, and Bonfire).*

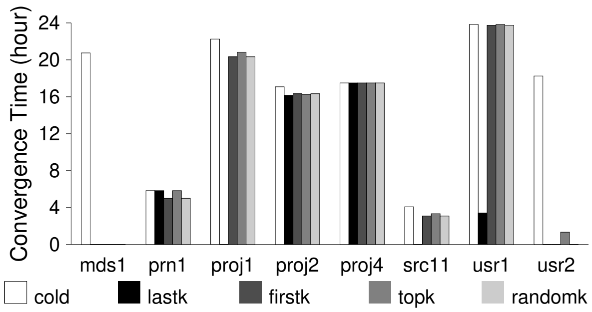


Figure 6: **Convergence Time**

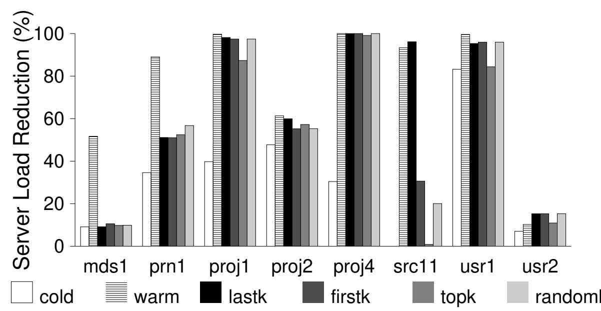


Figure 7: **Server I/O Load Reduction**

prefetching size. With prefetching, the convergence time and the server load reduction of the cold cache is better than those without prefetching. We find that disk prefetching lowers the benefit of cache warmup in some cases, but cache warmup still provides an improvement of up to 62% in convergence time and up to 228% in server load reduction over the cold cache. We do not present the detailed prefetching results due to space constraints.

4.3.3 Algorithm Comparison

We now study the effect of different algorithms on different traces. In this calculation, we fix cache size to 50% WSS, warmup size to 10% WSS, violation threshold Th_V to 80%, convergence threshold Th_N to 3%, and warmup region size to 1 MB, a reasonable and common cache setting that uses parameter values in the middle of the parameter value range.

Convergence time: We plot the convergence time of day five of each trace with a cold cache and with caches warmed by different algorithms in Figure 6. We find that all algorithms improve or are the same as cold caches for all traces. Last-K is the best algorithm in terms of convergence time (small is better) for most traces and Top-K performs the worst. For example, for *mds1*, *proj1*, *src11*, and *usr2* traces, the Last-K algorithm reduces the convergence time to zero. These traces have a significant amount of hourly reaccesses. For the *web2* trace, the convergence time of the cold cache is zero on day five; therefore no improvement is possible for any algorithm.

Server Load Reduction: We then study the server load reduction and plot the reduction percentage of day five of each trace with a cold cache, an always-warm cache, and caches warmed up by different algorithms in Figure 7. Server load reduction is measured over the convergence time period of the cold cache for all schemes. We find that all algorithms reduce more server load than the cold cache (bigger is better). In some cases, Bonfire algorithms outperform the always-warm cache. Since Bonfire uses a 1 MB warmup region size and the cache uses 4 KB block size, Bonfire has the effect similar to prefetching. Even in terms of server load reduction, we find Last-K to be the best algorithm in general.

Daily Reaccesses: In Figures 6 and 7, we use the original trace days which start at 5PM every day. However, the daily reaccesses happen at other times. To evaluate the effect of daily reaccesses, we shift the daily reac-

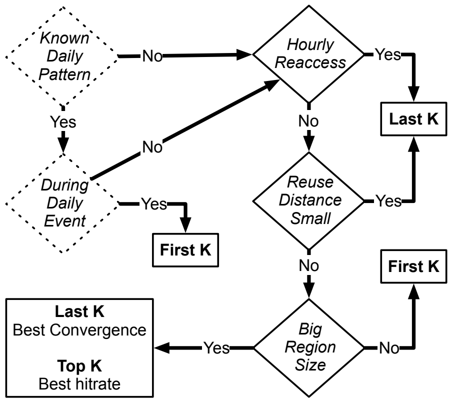


Figure 8: **Algorithm Decision Tree** *Dotted boxes imply that wall clock time needs to be taken into consideration.*

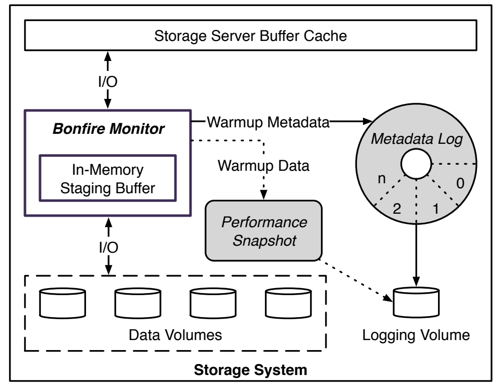


Figure 9: **Bonfire Architecture** *The warmup metadata flows are shown with solid lines and data flows with dotted lines.*

cessed traces so that the per-day traces start right before the daily reaccess events. We find that First-K is the best for daily reaccesses. They have the same convergence time as Last-K; the server load reduction of First-K is 110% while that of Last-K is 87%.

Algorithm Decision Tree: We use the simulation results for various parameter settings and correlate them with workload behavior in Section 3 to formulate a decision tree (Figure 8) to choose the best performing algorithm based on workload properties. Overall, we find that Last-K works well for most traces. For traces with a known daily pattern, First-K works best when the cache is started just before the recurring daily event. For traces with hourly reaccesses, Last-K works the best. It is also the best algorithm for traces with other reaccesses with small reuse distance. When reuse distance is big, First-K works well when the warmup region size is small and Last-K and Top-K work well when the region size is big; these cases are rare in the traces.

4.4 Key Findings

1. Cache warmup is effective in both improving convergence time and server load reduction.
2. Last-K is the best algorithm in general and First-K works better for known recurring daily patterns.

5 Bonfire Architecture

The Bonfire architecture is guided by three design principles: maintaining low overhead during normal operation, providing fast warmup when starting a new cache, and using a general design that is broadly applicable to a range of storage-level cache warmup scenarios. Sections 3 and 4 show that the Last-K algorithm provides the best benefit overall across a variety of traces. Hence, Bonfire uses the Last-K algorithm.

Figure 9 presents the overall Bonfire architecture. Bonfire runs in the storage server below its buffer cache; it monitors all block I/Os sent from the storage system buffer cache to the storage data volumes. It consists of two components: a small in-memory staging buffer and a local logging volume (e.g., a volume on a hard disk or on an SSD). Bonfire works in two phases: a *monitoring phase* where Bonfire tracks the recently used (Last-K) blocks of data for future cache warmup, and a *warmup pre-load phase* when Bonfire uses the monitored information to efficiently load the warmup data blocks into the cache. Bonfire uses two key insights in its design. It leverages the storage system buffer cache to provide a list of recently accessed blocks (Last-K). To achieve fast preloading, Bonfire uses the *performance snapshot* to layout warmup data on the logging volume such that it can be read sequentially to provide faster warmup. We explain the details of each phase in more detail next.

5.1 Monitoring Phase

Bonfire monitors the recently accessed blocks (Last-K) by logging the block I/O sent from the storage system buffer cache to the backend data storage. This approach is advantageous in several ways. The buffer cache maintains its contents using an LRU-like replacement policy; the stream of buffer cache I/Os are thus suited to track a temporal pattern. With this insight, Bonfire does not need to maintain additional data structures. In addition, Bonfire is not in the performance-critical path as it monitors buffer cache loads; a properly configured system has few cache loads and cache loads already incur a disk read penalty. Another optimization Bonfire performs is to stage the warmup metadata in an in-memory staging buffer before flushing to a logging volume; the staging allows for large sequential writes to the logging volume.

By only recording warmup metadata (*metadata-only*), one has to read the block contents from the underlying disks during warmup pre-loading. However, the original block contents may be scattered randomly on the backend data disks resulting in many random I/Os. To reduce pre-load time, Bonfire can write warmup data to a *performance snapshot* in the logging volume in addition to the warmup metadata (*metadata+data*). Performance snapshots trade off faster warmup pre-loading time for increased logging volume space and bandwidth. Specifically, we store the data read and written below the server buffer cache in an in-memory data staging buffer and flush it to the performance snapshot when it is full. The latest data is written to the end of the performance snapshot; during warmup only the latest data is used.

5.2 Warmup Pre-load Phase

Bonfire pre-loads a new cache using its warmup logs for a user specified warmup data size, e.g., 25% of the size of the new cache. The cache warmup proceeds in one of two ways depending on what information was logged during the monitoring phase.

With a *metadata-only* log, Bonfire reads the circular metadata log from the most recently written metadata to the least recently written ones. After all the metadata is read into memory, it adds the contents of the staging metadata buffer. After sorting all the metadata and removing duplicate LBAs, Bonfire reads the warmup data from original data disks and writes them into the new cache.

With a *metadata+data* log, Bonfire first loads the warmup data in the staging buffer to the new cache. It then reads each chunk of the metadata log and the corresponding data log on the logging volume into memory in reverse time order; most recently written data is read first. If a block LBA has not been seen before, Bonfire writes it to the new cache. Since most recently written logs are read first and loaded into the new cache first, we make sure that no stale data is written in the new cache.

If during warmup the staging buffer content is unavailable, we only use the warmup metadata and data from the logging volume. If the logging volume is unavailable, the warmup process falls back to the on-demand warmup.

6 Evaluation

In this section, we describe our implementation of the Bonfire prototype and present our experimental evaluation. Our goal is to measure the benefit of Bonfire in reducing cache warmup time, server load, client perceived read performance, and to understand the cost associated with these improvements.

We implement the Bonfire prototype as a user space trace-replay program. As the MSR-Cambridge traces are captured below the storage server's buffer cache, replaying them accurately simulates the I/O stream from buffer cache to the backend data disks [19]. Our trace replay issues synchronous I/Os and ignores request time, allowing our experiments to finish faster and be independent of the hardware environment. We implement both the logging schemes of Bonfire: *metadata-only* and *metadata+data*. In our experiments, we first load the cache warmup data into the new cache and then let the foreground workloads use the warmed cache.

We compare Bonfire with the warm cache that has been running from the beginning of the traces (*always-warm*) and the cold cache which is warmed up on demand after the new cache starts (*cold*).

Experimental environment: Experiments were conducted on a 64-bit Linux 3.3.4 server, which uses a 2.67 GHz Intel Xeon X5650 processor and 32 GB of RAM. The data storage consists of three concatenated 500 GB 7200 rpm SATA hard drives. We use a separate hard disk as the logging volume. The storage-level cache in all our experiments is a 256 GB DRAM. Since all our experiments use synthetic workloads and trace replays, they are not dependent on the content of the datasets. Thus, we allocate a small in-memory buffer as the cache to read or write data with random content. With this approach, we can measure the overhead of accessing the DRAM cache and also emulate large storage-level cache sizes.

6.1 Synthetic Workload

We first evaluate our Bonfire prototype using a synthetic Zipf workload [9], which provides a controlled way to examine the effect of different cache warmup schemes. We generate a trace of five million 4 KB reads with the Zipf distribution, using $\alpha = 0.4$ and $N = 50\,GB$; its working set size is 4.8 GB. We then split the trace equally into two sub-traces. In the first half, no storage-level cache is used for the *cold* or the Bonfire schemes but the *always-warm* scheme begins to use the storage-level cache. For the cold and the Bonfire schemes, we trigger a new storage-level cache creation and initiate warmup for Bonfire at the end of the first half.

Reduction in Server Load: Figure 10 presents the read hit-rate against the number of I/Os for *cold*, *always-warm*, and Bonfire; higher read hit-rate imply less server load. We vary the amount of warmup data fetched by Bonfire from 25% to 100% of the working-set size. The *cold* scheme fetches the blocks on demand after the creation of the new cache at the 2.5M I/O mark, resulting in it achieving a hit rate of 84% at the end of the trace. *Cold* falls behind and does not reach the hit-rate of the *always-warm* scheme (96%) by the end of the trace, using a Th_V of 10% and Th_N of 3%. In contrast, Bonfire fetches the data in bulk and pre-loads the cache. If the warmup size is limited to 25%, Bonfire achieves a hit-rate above 90% of *always-warm* at the 4M I/O mark. Larger warmup enables faster hit-rate convergence; Bonfire with 50% warmup data converges at the 3.6M mark I/O mark; a 75% warmup size allows Bonfire to converge at the 2.8M I/O mark. Bonfire with a 100% warmup size converges immediately at the 2.5M I/O mark.

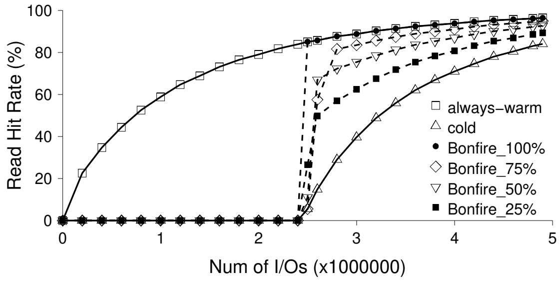


Figure 10: **Read hit rate of the Zipf workload**

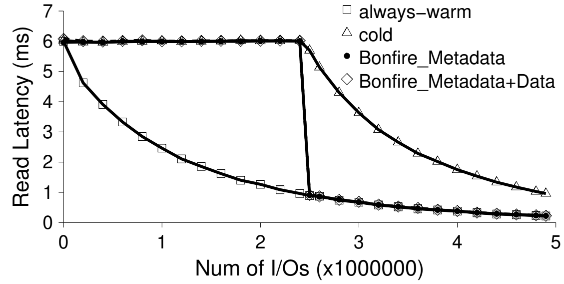


Figure 11: **Read latency of the Zipf workload**

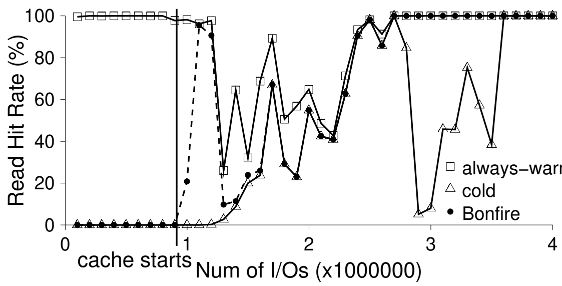


Figure 12: **Read hit rate of Day 5 and 6 of *proj1***

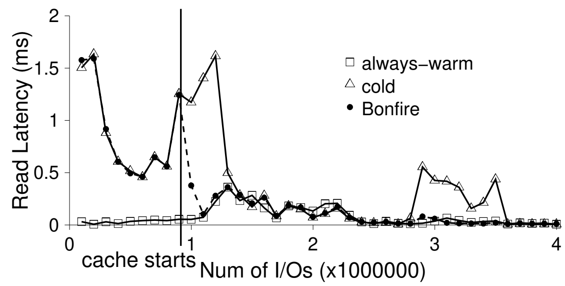


Figure 13: **Read latency of Day 5 and 6 of *proj1***

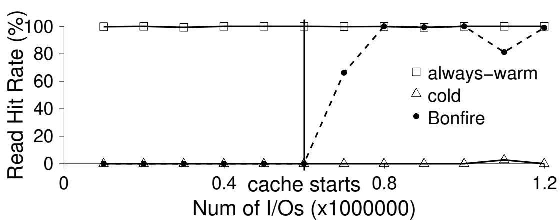


Figure 14: **Read hit rate of Day 5 and 6 of *web2***

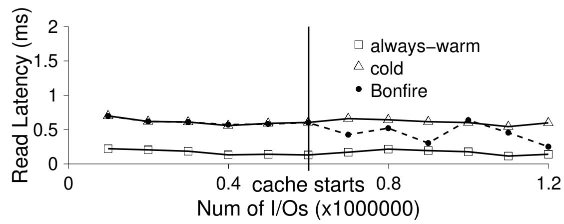


Figure 15: **Read latency of Day 5 and 6 of *web2***

Improvement in Application Performance: We also compare the read latencies of the *cold*, the *always-warm*, and the Bonfire warmed cache. Figure 11 plots the read latency over the number of I/Os. Bonfire achieves performance equal to the *always-warm* scheme (averaging at 0.5 ms) after the new cache starts, while the *cold* scheme averages at 2.6 ms.

6.2 Trace-Based Workloads

Next, we examine Bonfire performance using the *proj1* and *web2* traces from the set of MSR-Cambridge traces. We select these two traces because they exhibit different properties. The *proj1* trace has more I/Os and a large working set but lower reaccess rate than the *web2* trace (see Table 1). From our analysis in Section 3, we also find that *proj1* exhibits mainly hourly reaccesses; the *web2* trace exhibits both daily reaccesses and reaccesses that are more than one day. The *proj1* trace also helps to stress Bonfire since it has a working set size (642 GB) that is larger than our storage-level cache size (256 GB). We replay day one to day six of the *proj1* and *web2* traces and enable the storage-level cache for *cold* and Bonfire at the beginning of day 6; the *always-warm* scheme uses the storage-level cache from day 1. The *proj1* and *web2* traces provide realistic workloads that can vary significantly over time; specifically, the hit-rate of *proj1* fluctuates wildly during day 6 and stabilizes at the end.

Reduction in Server Load: We examine the read hit-rate to determine how Bonfire reduces storage server load. Figure 12 presents the read hit rates of *always-warm*, *cold*, and Bonfire over number of I/Os with the *proj1* trace. We find that the *cold* cache does not converge with the *always-warm* cache until around 2.7M I/Os after the new cache starts (at around 0.9M I/Os). In contrast, Bonfire converges around 1.1M I/Os after the new cache starts (a 59% improvement). Figure 12 also shows two interesting results. First, we see that with Bonfire the read hit rate immediately increases when the new cache starts while with the cold scheme, the read hit rate increases gradually. However, the workload fluctuations cause the hit-rate to drop thereby delaying convergence time for Bonfire until the 2.1M I/O mark. Second, after Bonfire converges, the hit rate improves and matches the hit-rate of the *always-warm* cache, while the hit rate of the cold scheme drops again at around the 3M I/O mark. The segment after 3M

	Zipf		*proj1*		*web2*	
	Metadata-Only	Metadata+Data	Metadata-Only	Metadata+Data	Metadata-Only	Metadata+Data
Log Vol Space	19 MB	9.5 GB	20 MB	9.9 GB	71 MB	36 GB
Avg Log Vol B/W	194 KB/s	97 MB/s	9.1 KB/s	4.6 MB/s	476 KB/s	238 MB/s
Pre-Warm Time	132 s	126 s	184 s	130 s	1224 s	664 s

Table 4: **Bonfire Overheads**

I/Os uses data that may have been accessed before the new cache is inserted. Both the always-warm and Bonfire would do well as they have captured the history, while the cold scheme will have more cache misses.

Similarly, for *web2*, Figure 14 presents the read hit rates of the three schemes. With Bonfire, the read hit rate increases right after the new cache is added and converges with always-warm after 0.2M I/Os. With the cold scheme, read hit rate stays around zero after the new cache starts.

Improvement in Application Performance: Figure 13 presents the latency results of the *proj1* trace. We see similar effect of Bonfire with respect to read latency as with read hit rate. The *cold* scheme has two areas with high read latency: one around the 1M I/O mark and another around the 3M I/O mark; these correspond with its low read hit rate regions. While Bonfire converges at the 2M I/O mark, the read latency before it converges is comparable with the latency of the *always-warm* scheme.

Figure 15 presents the latency results of the *web2* trace. Similar to its read hit rate results, we see that with Bonfire, the read latency drops after the new cache starts. The read latency of *cold* stays around 0.6 ms. The *web2* trace contains mostly sequential I/Os; even though the read hit rate is low for the *cold* scheme, reading data from backend disks is reasonably fast.

6.3 Overhead Analysis

We quantify the Bonfire overhead in terms of fixed and variable costs. The fixed costs include the metadata and the data staging buffer, whose sizes can be configured by user. Bonfire also imposes a variable cost based on the workload. Logging warmup metadata and data adds bandwidth and storage space overhead; pre-loading warmup data to the new cache extends the time when the foreground workload can start to use the new cache.

Table 4 presents the overhead of the Zipf workload, the *proj1* trace, and the *web2* trace with the *metadata-only* and the *metadata+data* logging schemes, when the logging metadata buffer size is 256 KB and the logging data buffer size is 128 MB. We measure the amount of logging data generated over time and use the original request access times in the *proj1* and *web2* traces to calculate logging bandwidth. For the Zipf workload, we assume an intense foreground I/O speed of 1 Gbps.

Overall, we find larger monitoring overhead and faster warmup data pre-loading with the *metadata+data* scheme than the *metadata-only* scheme. For the Zipf workload, we see a high demand for logging volume bandwidth when using the *metadata+data* scheme, averaging at 97 MB/s. The benefit of *metadata+data* scheme on warmup data loading time is small. The Zipf workload is drawn from a 50 GB logical address space and has a small working set; reading the warmup data from backend data disks is not much slower than reading it sequentially from the performance snapshot. For the *proj1* and *web2* traces, the *metadata+data* offers more benefit, improving warmup loading time by 29% and 46%. Both traces have larger logical address spaces than the Zipf workload; warmup data is more likely to be random and reading it from backend disks is slower. However, for *web2*, *metadata+data* has a high logging bandwidth. The *metadata+data* scheme logs all foreground I/Os; its logging bandwidth is thus dependent on the workload I/O bandwidth. The *web2* trace has intensive I/Os, resulting in its high *metadata+data* overhead. Therefore, when there are more idle times in the I/O requests and when the data logical address space is large, the *metadata+data* scheme may offer more benefit for a fast cache warmup.

Next, we study the trace completion time of different schemes. For the Zipf workload, the *cold* scheme runs for 1.8 hours while Bonfire using the *metadata-only* scheme and Bonfire using the *metadata+data* scheme both complete in 23 minutes, including pre-load time. For the *proj1* trace, the *cold* scheme runs for 13 minutes while Bonfire using the *metadata-only* scheme and the *metadata+data* scheme completes in 9.5 minutes and 12 minutes. We find that for both the Zipf workload and the *proj1* trace, the cost of Bonfire pre-loading plus the foreground workload run time is still smaller than the on-demand warmup time, suggesting that Bonfire uses the backend disks more efficiently leading to the performance improvement.

For the *web2* trace, the *cold* scheme runs for 6.5 minutes; Bonfire using the *metadata-only* scheme and the *metadata+data* scheme completes in 25 minutes and 19 minutes. The read latency of *cold* with the *web2* trace is much lower than with the Zipf workload and the *proj1* trace because of its I/O sequentiality. However, the *web2* trace requires a high overhead of Bonfire logging and pre-loading (see Table 4), resulting in Bonfire running longer than the *cold* scheme. To reduce the Bonfire overhead, the workload idle time can be utilized to perform logging and pre-loading; flash-based SSDs can also be used as the logging volume.

Finally, we study the effect of the staging buffer size and vary the metadata buffer size from 64 KB to 1 MB. The data buffer size is proportional to the metadata one and vary from 32 MB to 512 MB. When the staging buffer is big, we find that less data is written to the performance snapshot; 512 MB data buffer size has up to 25% reduction in total warmup data written as compared to the 32 MB size. With bigger staging buffer, more overwrites can be absorbed in the buffer. During warmup time, Bonfire also benefit from a bigger staging buffer; less data needs to be read from the smaller performance snapshot and a larger part of the warmup data is still in the memory staging buffer. With 512 MB data buffer, warmup time improves up to 50% as compared to the 32 MB one. Overall, we find that larger staging buffer benefit both the monitoring and the warmup phase. However, a big staging buffer consumes more memory. Moreover, if the staging buffer is unavailable during warmup, more warmup data is lost with bigger staging buffer. Therefore, we choose to use 256 KB metadata and 128 MB data staging buffer.

6.4 Summary

Our evaluation of the Bonfire prototype with a synthetic workload and two real traces shows that Bonfire largely reduces the cache warmup time over a cold cache that is warmed up on demand and increases cache read hits. These benefits come with a small fixed monitoring memory overhead and a small logging space and bandwidth overhead when using the *metadata-only* scheme. The *metadata+data* scheme has a higher logging overhead, especially when there is little I/O idle time, but allows faster warmup data pre-loading time.

7 Related Work

We present the related work including workload studies, storage-level cache usage, and existing approaches to fast cache warmup.

Workload studies: Workload studies can provide valuable data needed to make informed design decisions. A number of workload studies have addressed caching to varying degrees. The most related is an analysis of distance between reaccesses by Zhou et al. [29]; their goal is to understand how it changes across multiple levels of cache to develop the Multi-Queue cache replacement algorithm. Adams et al. [3] focused on long-term behavior, over a coarser time-frame. Ellard et al. [11] examined NFS traces and examined caching as it pertains to block lifetimes. Roselli et al. [25] focused on caching with regard to disk performance. Recently, Harter et al. [17] studied the I/O behavior of Apple desktop applications.

Storage-level cache: Storage-level caches are becoming common in modern storage systems. Memcached is a distributed memory caching system that has been used widely by different systems [13]. Suggestions have been proposed to use application-level scripts to pre-populate important data to warm the cache [1, 2]. Similarly, Amazon ElastiCache provides distributed in-memory caches in the cloud, and relies on redundancy to reduce the effect of warmup times [5].

RAMCloud [23] is a storage architecture that stores all data in DRAM to consistently achieve low-latency access. RAMCloud maintains disk or flash-based backups that are used for recovering from failures [23]. Bonfire-like cache-warmup approaches may still apply in these scenarios.

The Rio file cache [10] enables memory contents to survive crashes, thereby reducing write traffic to storage. However, Rio does not handle restarts with loss of power or those involving hardware upgrades. Further, warmup scenarios such as dynamic cache creation would not benefit from Rio.

Cache warmup techniques: Windows SuperFetch is a technique to reduce system boot time and application launch time by pre-loading commonly used boot sequence and applications into memory based on history usage pattern. Bonfire is similar to SuperFetch in that it also preloads warmup data and uses the access pattern in history. However, Bonfire is different from SuperFetch in several ways. First, Bonfire monitors and warms up storage-level caches, which is much bigger than traditional buffer caches. Second, Bonfire monitors all workloads below the server buffer cache instead of at the application level. Finally, Bonfire can choose to use the performance snapshot to further improve the warmup time.

8 Conclusion

Large caches are becoming the norm in the storage stack. When they are empty, they serve little purpose. Unfortunately, a combination of scale and dynamism in today's data centers is causing caches to be empty more often. Therefore, the process of warming caches needs to be studied in detail.

As a first step, we analyze server-side workload traces and quantify key workload characteristics that can be used to design warmup mechanisms. We propose Bonfire to effectively and efficiently bulk-load storage-level caches with useful data. We find from experiments that Bonfire reduces the time needed for the cache to reach the performance level of an "always-warm" cache. Further, Bonfire greatly reduces the load on the backend storage. The runtime overhead due to Bonfire is also small, thus making it viable for a large number of workloads.

We believe that the techniques we use in Bonfire can also be extended to other cache warmup scenarios, such as client-side cache warmup and virtual machine restarts; we leave it as our future work.

Acknowledgments

We thank the anonymous reviewers and Eno Thereska (our shepherd) for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. We also thank the members of the NetApp Advanced Technology Group and the ADSL research group for their insightful comments.

References

[1] A Rant about Proper Memcache Usage. `http://joped.com/2009/03/a-rant-about-proper-memcache-usage/`.

[2] Memcached Warmup Scripts. `http://www.pablowe.net/2008/07/memcached/`.

[3] I. F. Adams, M. W. Storer, and E. L. Miller. Analysis of Workload Behavior in Scientific and Historical Long-Term Data Repositories. *ACM Transactions on Storage (TOS)*, 8(2), May 2012.

[4] Amazon. Amazon EC2 Service Level Agreement. `http://aws.amazon.com/ec2-sla`.

[5] Amazon. Amazon ElastiCache. `http://aws.amazon.com/elasticache/`.

[6] Amazon. Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3/.

[7] L. N. Bairavasundaram, G. Soundararajan, V. Mathur, K. Voruganti, and K. Srinivasan. Responding Rapidly to Service Level Violations Using Virtual Appliances. *ACM SIGOPS Operating Systems Review*, 46(3):32–40, December 2012.

[8] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.

[9] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 1999)*, New York, New York, March 1999.

[10] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, Cambridge, Massachusetts, October 1996.

[11] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[12] EMC. EMC Symmetric VMAX 20K Storage System. `http://www.emc.com/collateral/hardware/data-sheet/h6193-symmetrix-vmax-20k-ds.pdf`.

[13] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5, August 2004.

[14] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[16] B. S. Gill. On Multi-level Exclusive Caching: Offline Optimality and Why Promotions Are Better than Demotions. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[17] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A Fiel is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[18] J. McHugh. Amazon S3: Architecting for Resiliency in the Face of Massive Load. In *The Annual International Software Development Conference (QCon '09)*, San Francisco, California, November 2009.

[19] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[20] D. Narayanan, A. Donnelly, and A. Rowstron. Private Conversation, September 2012.

[21] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[22] NetApp. FAS6200 Series Technical Specifications. `http://www.netapp.com/us/products/storage-systems/fas6200/fas6200-tech-specs.html`.

[23] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[24] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, Colorado, December 1995.

[25] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, San Diego, California, June 2000.

[26] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: a Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the EuroSys Conference (EuroSys '12)*, Bern, Switzerland, April 2012.

[27] M. Wachs and M. Abd-El-Malek. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, San Jose, California, February 2007.

[28] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[29] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.