The Design and Implementation of RAPID-Cache: A Reliable, Inexpensive, and High-Performance I/O Cache for Linux

Zhiyong Xu and Yiming Hu
Department of Electrical & Computer Engineering and Computer Science
University of Cincinnati
email: {zxu,yhu}@ececs.uc.edu

Abstract

Most file system performance enhancing techniques, such as the I/O buffer cache and the Log-structured File Systems (LFS), relying on caching data in volatile RAM for a period of time before destaging them to disk. Such approaches pose a reliability problem. Non-volatile RAM (NVRAM) caches can be used to enhance the reliability. However, large NVRAM caches are very expensive and not as reliable as disks.

This paper presents the design and implementation of a different approach called RAPID-Cache (Redundant, Asymmetrically Parallel and Inexpensive Disk Cache) for highly-reliable systems. The new approach provides an inexpensive way to make the OS buffer cache non-volatile and highly reliable. Recent researches using simulation suggest that as a cache inside a disk array controller, RAPID-Cache has great reliability/cost advantages over conventional write cache structures. In order to validate whether RAPID-Cache can really achieve satisfactory performance in real systems, and to see if RAPID-Cache can also improve the performance/reliability of the OS buffer cache, we have designed and implemented RAPID-Cache for the Linux system. Measurement results using realistic benchmarks show very promising results. Using only 3.25 MB of NVRAM, a RAPID-Cache outperforms normal NVRAM caches of 64 MB or even 128 MB. RAPID-Cache also provide better reliability than normal NVRAM caches.

1 Introduction

Disk I/Os are a major performance bottleneck for many applications. Many solutions have been proposed to improve disk I/O performance, such as the file system buffer cache used in most modern OSs, the Log-structured File Systems (LFS) [1], Meta-data Logging [2], Soft Updates [3] and Disk Caching Disks (DCD) [4]. Most of these techniques keep some dirty data in RAM for a period of time before destaging them to disk. For example, in most Unix systems, dirty data are left in the buffer cache for 30–60 seconds before they are written to disk. Such a delay between updating data in RAM and committing them to disk poses a potential reliability problem.

Some servers use non-volatile RAM (NVRAM) to improve both performance and reliability. NVRAM is normally used as a write cache to reduce the I/O response time seen by users [5, 6]. In case of a system failure, the updated data is still in NVRAM and can be recovered later. Treiber and Menon [6] pointed out that the use of NVRAM can reduce disk utilization for writes by an order of magnitude. Because of these, NVRAM plays an important role in high performance computer systems. However, the use of an NVRAM write cache introduces two problems: (1) it still has lower reliability compared to disks, and (2) it has a very high cost.

Disks are impressively reliable today, with a Mean Time To Failure (MTTF) of up to 1 million hours. Such a low failure rate, coupled with possible redundancy such as RAID (Redundant Array of Independent Disks) systems used by many servers, gives a Mean Time To Data Loss (MTTDL) of several hundreds of millions of hours in a typical RAID-5 system [7]. Adding a single NVRAM write cache in front of a disk system creates a reliability mismatch and a single point of failure, which is vulnerable to data loss. Savage and Wilkes pointed out in [7] that because typical NVRAM technology (battery backed RAM) has a quite low MTTF of 15K hours. A single-copy NVRAM cache suffers significantly higher risk of data loss than results from disk failures. To overcome the reliability problem, it is possible to use dual-copy caches so that a failure in one cache leaves the other cache intact [8]. When a write request comes, the controller writes two copies of the data independently into the two caches, a primary cache and a backup cache.

Besides the reliability problem, NVRAM is also known to be very costly [7, 9] so the size of the NVRAM cache is often limited. For example, a major NVRAM manufacturer quoted the price of NVRAM with embedded lithium-cell batteries for $50/MB in quantity recently. The cost of disks, on the other hand, is about 0.1 cents/MB, which is a difference of four orders of magnitude. Moreover, the cost difference is widening (the difference was three orders of magnitudes two years ago), because prices of disks are falling very rapidly. For a system with a reasonably sized write cache (say 128 MB or more), the cost of NVRAM is much higher than those of disks, and will dominate the cost of the entire system. If we use dual-copy caches to ease the reliability problem of the single-copy cache, the cost becomes prohibitively high, particularly for large caches.

1.1 RAPID-Caches

The standard dual-copy write cache system has a symmetric structure, where both the primary write cache and the backup
write cache have the same size and the same access characteristics — fast read speed and fast write speed. However, the backup cache does not provide any performance benefit to the system during normal operations. Therefore it is wasteful to use a backup cache identical to the primary cache. What is needed is only a backup cache that can be written to very quickly while its read operations are not as critical, since reads from the backup cache occur only during error-recovering periods.

Based on these observation, Hu and Yang proposed a new disk cache architecture: Redundant Asymmetrically Parallel and Inexpensive Disk Cache or RAPID-Cache for short [10, 11]. It uses an asymmetric structure: a conventional, fast-write-fast-read cache as a primary cache and a non-volatile, fast-write-slow-read cache as a backup cache. The primary cache is made of normal NVRAM or DRAM, while the backup cache consists of a small NVRAM cache and a cache disk (or disk partition). In the backup cache, small and random writes are first buffered in NVRAM, then they are assembled into some large logs and written to the cache disk later, which is similar to Log-Structured File systems [1, 12]. This two-level backup cache appears as a large NVRAM and can achieve nearly the same write speed as the primary cache. The slow read speed of the cache disk is not a concern since normal read requests go to the primary cache. Using RAPID-Cache, we can get a system with higher reliability compared to one-copy write buffer, and a lower cost compared to dual-copy write buffer.

1.2 Using RAPID-Caches to Improve OS Buffer Cache Performance and Reliability

In [10, 11], trace-driven simulation experiments show that RAPID-Caches are very effective as write caches of large RAID systems. We try to apply the same idea to a different system component: the Unix I/O buffer cache. The basic idea is to use a reliable, inexpensive, fast-write-slow-read backup cache (consists of a small NVRAM buffer and a cache disk) to supplement the regular RAM-based I/O buffer cache (which acts as the primary cache). During normal operations, the system writes two copies of the same data, one to the buffer cache (primary), the other to the backup cache. Most data blocks in the backup cache are kept in the cache disk, which is more reliable than NVRAM. If one copy of data is corrupted because of hardware, power, and/or software failures, the other copy can be used to recover the data. Note that such a configuration is particularly resilient to data corruption caused by buggy software. “Wild-writes” caused by buggy software can easily cause random data corruption in RAM. On the other hand, to access data on disk and in most NVRAM systems, a well-defined sequence of actions must be followed. It is unlikely that run-away bugs will following such sequences.

This paper tries to address the following two questions:

1. Is the RAPID-Cache architecture suitable for the OS I/O buffer cache?

The original RAPID-Cache architecture was designed for I/O caches of large RAID systems. It takes I/O requests after they are filtered by the OS buffer cache. When integrating the RAPID-Cache into the OS buffer cache, it is not clear how the architecture will interact with other kernel components.

2. What is the real performance of RAPID-Caches in a real system?

The performance results (I/O response times and throughput) in the original studies of RAPID-Caches were obtained using simulation only. However, [13] have shown that I/O response times are not always a good performance indicator in I/O systems because not all I/O requests affect the system behavior in the same way. Trace-driven simulations also have limited accuracy because there is no feedback between the traces and the simulators. The main objective of this research is to examine whether RAPID-Cache will really have good performance in the real world. We believe that the best way to evaluate the real world performance is to actually implement the RAPID-Cache architecture and measure its performance in terms of program execution times, using some realistic benchmarks.

The paper is organized as follows. Section 2 presents the overview of RAPID-Cache, Section 3 gives detailed description of data structures used in RAPID-Cache implementation. In Section 4, we describe the operations in RAPID-Cache. Section 5 includes performance testing results. We discuss related work in Section 6 and conclude the paper in Section 7.

2 Redundant Asymmetrically Parallel and Inexpensive Disk Cache (RAPID)

Figure 1 shows the general structure of RAPID-Cache. It consists of a conventional primary RAM cache and a backup cache. The backup cache is a two-level hierarchy with a small NVRAM on top of a cache disk, similar to DCD [4].
composed of NVRAM and a cache disk, both of them are non-volatile, all dirty data in primary cache have copies in backup cache. In case of system crashes, no data loss occurs.

Although disk access speed is much slower compared to DRAM and NVRAM, the two layered structure of backup cache integrates several small writes into one big disk write, which greatly reduces seeking and rotation time. So these additional disk writes to cache disk will not degrade system performance too much.

RAPID-Cache does need a cache disk, which may incur an additional cost. However, the cache disk size can be small, since the amount of valid data in the entire backup system is the same as those in the primary cache. We find that to achieve good performance, the cache disk size should be around 4–5 times of the primary cache size. For example, if the primary buffer cache size is 128MB, then a cache disk of 512 MB is sufficient. Larger disk is not necessary, while too smaller a disk may cause too much garbage collection traffic (to be discussed shortly). As of this writing, an EIDE disk of 30 GB has a retail price of about $50, which is fairly inexpensive comparing to that of NVRAM ($50/MB). Moreover, since we need only 512 MB (less than 2% of the total 30 GB capacity) of space to cache data, the remaining space can be used to store other files, as long as these files are not accessed too frequently to cause much increased traffic to the disk. To reduce cost, in a multi-disk system, instead of buying a new disk, we can also choose a less-busy disk and create a cache disk partition on it.

2.1 Writes

In RAPID-Cache, when a write request comes, the system sends the data to both the primary cache and the backup cache. In the primary cache, if there’s no more space available, the operating system will destage some dirty blocks to the data disk and make room for the new coming data. Meanwhile, the copy of the destaged data in the backup cache is also invalidated.

2.2 Reads

Read is straightforward in RAPID-Cache. All read requests are sent to the primary cache, just as the cases in a regular system. The backup cache is not involved in read operations except in crash recovery.

2.3 Garbage Collection

Log-based storage solutions, such as LFS or DCD, have to perform garbage collection or data destage from time to time in order to free the log space. Garbage collection or destaging process can significantly reduce the overall system performance when the workload is high.

RAPID-Cache also needs garbage collection to free up cache disk space. However, the operation itself is much easier than in LFS. Since all data on the cache disk have a copy in the primary cache, to do garbage collection, the RAPID Cache simply chooses several fragment segments, copies the corresponding data from the primary cache to a segment buffer and write it to the cache disk, and invalidate all these fragment segments. Only one segment write is invoked and no disk read is needed. The overhead is much less than in LFS (LFS needs to read several segments from disk).

In addition to the low-cost garbage collection algorithm, in actual experiments, we found that for the workload we used, the RAPID-Cache almost never had to call the garbage collector, meaning that the garbage collection overhead has virtually no impact on the overall system performance. This is due to the following two reasons. First, because disk spaces are so inexpensive now (about 0.1 cents per MB as of this writing), we normally choose a cache space that is 5–10 times larger than the primary write cache size. For example, for a primary buffer cache size of 128 MB, we can use a disk space of 512 MB as the cache space. Since the 128 MB of data are spread over a space of 512 MB, much of the disk space is empty (or invalid) therefore does not need garbage collection. Second, unlike an LFS system which writes both actively-accessed data and inactive data into a segment, in a RAPID-Cache system, active data and inactive data are separated. Most active data are kept in the LRU cache, while data in the disk segments of the cache-disk are relatively inactive. Therefore entire segments are often invalidated because the background destage threads constantly destage inactive data to disk. As a result, most of time the system can find an empty disk segment without the need for garbage collection.

2.4 System Recovery

If one copy of data is corrupted because of hardware and/or software failures, the other copy can be used to recover the data. In case of power failure, dirty data in the primary cache (RAM) will be lost. The backup cache will be used to recover the data after reboot.

3 Design and Implementation

In this section, we describe the design and implementation of RAPID-Cache for Linux. The experimental system is built on Redhat Linux 7.2 with an upgraded 2.4.12 kernel. The RAPID-Cache implementation is integrated into the kernel with moderate modification to the file system source code. In the modified kernel, each dirty block has copies in both the system buffer cache and the RAPID-Cache backup cache.

Linux makes intensive use of the buffer cache to reduce the overhead of slow disk I/Os. Each block in the buffer cache corresponds to one data block on disk and has a buffer head storing its description. The kernel manages these blocks by buffer heads. Each buffer must stay in one of the following lists: Unused Buffer List, Locked Buffer List, Updated Buffer List or Dirty Buffer List. Dirty Buffer List maintains all dirty blocks before the modifications arrive the data disk [15, 16, 17] and form the primary write cache. RAPID-Cache replicates all these dirty blocks in the backup cache and eliminates the possibility of information loss.

Figure 2 shows the general structure of the RAPID backup cache. It includes three main parts: a small size of NVRAM, a small portion of system memory and a cache disk or a disk partition. Data structures in NVRAM include (1) a LRU Cache which backups the most recently written data blocks, (2) two Segment Buffers which buffer several data blocks before writing them to
the cache disk, (3) a Segment Table maintaining the usage information of segment buffers and all cache disk segments, and (4) a LRU Cache Table storing information of all valid blocks in the LRU Cache. These are the most important data structures and must be kept in NVRAM. In case of a crash, they are used to rebuild data. Data structures in the system memory include (1) a Rapid LRU List which is a doubly-linked list connecting all blocks in the LRU Cache, (2) a Free Rapid Cache Head List maintaining all the free rapid cache heads, and (3) a Data Lookup Table. Our design is sightly different from the original RAPID-Cache in utilizing system memory management for data lookup table. These reason is that the lookup table is accessed very frequently therefore its performance is important. For the NVRAM card we used, its access speed is slower than that of system DRAM because each access has to go to the PCI bus. The table content can be rebuilt after a crash from data on the cache disk and in the NVRAM.

### 3.1 Cache Disk Organization

The cache disk is the lower level storage device of RAPID-Cache, only the relatively inactive data blocks evicted from the LRU Cache are stored on it. Figure 3 shows the structure of the cache disk. It is divided into several large, fixed-length segments, similar to the cases in LFS and DCD. Each segment contains a segment head followed by a fixed number of slots (4 KB each), which is the same as described in segment buffers.

The segment size in our implementation is 128 KB, which is the maximum I/O size in Linux. The 128 KB segment contains a segment head and 31 slots. A segment is the basic unit for disk I/O in RAPID-Cache. The RAPID backup cache reduces disk I/O overhead by converting many small disk writes into several large segments I/Os, therefore significantly reduce time spent on disk seeking and rotation.

![Cache Disk Organization](image)

### 3.2 NVRAM data structures

#### 3.2.1 LRU Cache

The LRU cache is the higher level of RAPID backup cache. It stores copies of the most recently written data in the primary buffer cache. When a block in the primary cache is modified and inserted into the Dirty Buffer List, this block is also sent to the backup cache. RAPID-Cache adds it to the LRU cache. If the LRU cache is full and has no place for this new block, several least recently used are destaged from LRU Cache to one segment buffer to make room for the incoming request. In case that block is already in the backup cache, the old copy will be invalidated. The detailed operation is given in Section 4.

The LRU cache is divided into many 4 KB slots. For a 2 MB LRU cache, the number of slots is 512. Each slot can store one data block. The description information of a valid block in LRU cache is stored in a Rapid cache head and each block has entries in both the Data Lookup Table and the Rapid LRU List. These data structures will be discussed shortly. The entry in Data Lookup Table speeds up searching for blocks. The entry in the Rapid LRU List is to keep the access order of all blocks in the LRU cache. These two lists have different coverage. Any valid data block in the backup cache, no matter in the LRU cache, a segment buffer or on the cache disk, has an entry in the Data Lookup Table. But only valid blocks in LRU cache have entries in the Rapid LRU List.

![2MB LRU Cache](image)

#### 3.2.2 Segment Buffers

Because the LRU cache is small, when it fills up, less active data are evicted and moved to the cache disk. Since a segment is the basic unit for disk I/O, data blocks evicted from the LRU cache are first assembled in one of the two segment buffers. When the buffer is full, the entire buffer is written in to an empty disk segment on the cache disk in a single large disk write. Since large I/O are very efficient, the write speed is very high. The segment buffers in NVRAM and the disk segments on cache disk have the same structure. The structure of a segment buffer is shown in Figure 5. It can be divided into two parts:

**Segment Head**: It stores information of all data blocks within this segment buffer. Its size is 2 KB and it contains the...
3.2.3 LRU Cache Table

LRU Cache Table is used to store information of all valid blocks in the LRU Cache. Since the block size is 4 KB, for a 2 MB LRU Cache, it contains 512 entries, each entry records rch_dev and rch_blocknr of a block. It needs \(512 \times (4+4)=4 \text{ KB} \) of storage space. And another 512 bits are used as a valid bit map.

<table>
<thead>
<tr>
<th>rch_dev</th>
<th>...</th>
<th>rch_dev</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>rch_blocknr</td>
<td>...</td>
<td>rch_blocknr</td>
<td>...</td>
</tr>
<tr>
<td>valid_bit 0</td>
<td>...</td>
<td>valid_bit 511</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 6: The Structure of the LRU Cache Table

3.2.4 Segment Table

The segment table records the current usage of all segment buffers in NVRAM and disk segments on the cache disk. Figure 7 shows segment table structure. Each entry in the table has the following fields:

- **valid_slot_num**: The number of valid slots in the segment (segment buffer or disk segment). The size is 4 bytes.
- **slot_LBA_table**: It has the same meaning as slot_LBA_table in a segment head. The size is \(31 \times (4+4) = 248 \text{ bytes} \).
- **valid_slot_bitmap**: This field describes which slots in this segment contain valid data blocks. It needs 31 bits for all slots. We use 4 bytes.
- **ptr_table**: It stores the pointers to both buffer head and rapid cache head of valid blocks in this segment. It has 31 entries. This table is used in garbage collection and the size is 248 bytes. Since the contents of this table can be rebuilt from other data structures after a crash, in our implementation this table is actually stored in DRAM, thus it does not take precious NVRAM space.

Figure 7: The Structure of Segment Table

A cache disk of 512 MB contains 4096 segments, plus two segment buffers in NVRAM. So the segment table has 4098 entries, each entry needs \(4+4+248 = 256 \text{ bytes} \). The total size of segment table is \(4098 \times 256 = 1 \text{ MB} \).

The Segment table along with the LRU cache table can identify all valid data blocks in the backup cache.

3.3 RAM data structures

3.3.1 Rapid Cache Head

Rapid cache head is the basic data structure in RAPID-Cache. Like a buffer head is used to identify a block in buffer cache, a rapid cache head uniquely identifies a valid block in the backup cache. Because each block in the backup cache is a replica of a dirty block in the primary cache, each rapid cache head corresponds to one buffer head in the primary buffer cache, and can identify one data block with maximum size of 4KB. It has similar structure as a buffer head. Figure 8 shows the organization of a rapid cache head.

Detailed fields descriptions are:
Rapid cache head

- \texttt{rch\_dev}: the real disk device number of the block. It is the same as \texttt{b\_dev} in the corresponding buffer head in the primary write cache.
- \texttt{rch\_blocknr}: the real block number of the block on \texttt{rch\_dev}, same as \texttt{b\_blocknr} in the corresponding buffer head.

The Logical Block Address (LBA) of a block in the backup cache is composed of \texttt{rch\_dev} and \texttt{rch\_blocknr}. It is used to locate a block in the \textit{Data Lookup Table}.

- \texttt{rch\_size}: the size of a block. It must be multiples of disk sector size (512 bytes). The maximum size is 4 KB.
- \texttt{rch\_prev} and \texttt{rch\_next}: pointers to the previous and next entries in one hash list in \textit{Data Lookup Table}. A double linked list with the same hash value is connected by these fields.
- \texttt{rch\_lru\_prev} and \texttt{rch\_lru\_next}: pointers to the previous and next entries in \textit{Rapid LRU list}, if a block has been moved to a segment buffer or cache disk, these two fields are set to NULL.

All rapid cache heads which point to blocks in the LRU cache are put in the \textit{RAPID LRU List}. Each time when a new block is inserted into the backup cache, we add it to the LRU cache first, and put its rapid cache head to the head of the Rapid LRU List. In case that the LRU cache is full, we only flush the entries in this list and cannot find this block (a cache miss), it means this data block cannot be read from the backup cache and written back to its corresponding location in the primary cache and on the data disk.

### 3.3.2 Data Lookup Table

One of the most challenging tasks in RAPID-Cache implementation is to design an efficient data structure and a searching algorithm for the data lookup table, which is used to locate data blocks in the backup cache. Each time when a block is inserted into the primary buffer cache, or a block in the primary cache is modified, RAPID-Cache must search the table to locate this block in the backup cache before adding it into the LRU cache. The data structures and searching algorithms must be simple and fast to reduce overhead.

While the Segment table along with the LRU cache table in NVRAM can identify all valid data blocks in the backup cache, they are not optimized for searching using block IDs. Moreover, compared to DRAM, NVRAM is expensive, and slower to access (because for PCI-based NVRAM cards, each memory access has to go to the slow PCI bus).

To solve this problem, an index called Data Lookup Table is created in RAM. The table can be reconstructed from data in NVRAM after a crash. Each rapid cache head has an \textit{entry} in the table, no matter it is in the LRU cache, in one of the segment buffers or on the cache disk. Like the buffer head uses \textit{device number} and \textit{block number} to address a block, we use \texttt{rch\_dev} and \texttt{rch\_blocknr} (LBA) as the search key for a rapid cache head in the data lookup table. Similar hash table and searching algorithms used in Linux are adopted here. Rapid cache heads which are mapped to the same hash value in data lookup table generate a doubly-linked hash list, which is connected through \texttt{rch\_hash\_prev} and \texttt{rch\_hash\_next}.

The detailed structure of the data lookup table is shown in Figure 9. Each time when a new block is added into the backup cache, RAPID-Cache locates the corresponding doubly-linked hash list according to its LBA value. Then this hash list will be searched from the head. If an entry’s LBA matches this new coming data block, it is a cache hit which means this new block is an updated version of a data block that already exists in the backup cache. This data block can be in the LRU cache, a segment buffer or on the cache disk, different operations will be done according to different situations. The detail is presented in Section 4. If RAPID-Cache reaches the last entry of this hash list and cannot find this block (a cache miss), it means this data block is added to the backup cache for the first time. We insert it to the LRU cache, and create entries for it in the data lookup table and the Rapid LRU list. In both cases, we add it to the head of the list.
3.3.3 Rapid LRU List and Free Rapid Cache Head List

A Rapid LRU List contains rapid cache heads of all valid data blocks in the LRU Cache. Rapid LRU List Head and Rapid LRU List Tail point to the start and the end of Rapid LRU List.

All unused rapid cache heads form another list – Free Rapid Cache Head List. Every time when a new data block is sent to the backup cache, RAPID-Cache allocates a free rapid cache head for it. Every time a data block is invalidated, its rapid cache head is freed and added to the Free Rapid Cache Head List again.

4 Operation Details

In this section, we describe operations in RAPID-Cache, which include write, read, garbage collection and crash recovery.

4.1 Writes

When a write request comes, the modified data block is sent to both the buffer cache (primary) and the backup cache. We did not modify original Linux operations on the primary buffer cache. RAPID-Cache is responsible for operations in the backup cache. First, the hash value of this data block’s LBA is calculated, then the corresponding hash list in the data lookup table is selected. If the hash list is not empty, RAPID-Cache starts searching from the head and stops when a rapid cache head with the same LBA is found (a cache hit). In this case, RAPID-Cache knows that the newly coming data block is an update to a block that already exists in the backup cache. The next operation is to check the location of this block in backup cache. It is done by checking rch_state field which must have one of the following three values:

1. **rch_state** is RCH_LRU_CACHE, which means that the data block is still in the LRU Cache. We simply overwrite its content in the LRU Cache, move the corresponding rapid cache head to the heads of both the hash list in the data lookup table and the rapid LRU list (which means it is the most recently accessed data). The operations are shown in Figure 10(a).

2. **rch_state** is RCH_LOCK, which means that the block is locked in one of the segment buffers for destaging to the cache disk. We must wait for the completion of this disk write, as shown in Figure 10(b). When the write finishes and the rch_state is changed to RCH_DISK, the following operations are the same as described next.

3. **rch_state** is RCH_DISK, which means that the block is on the cache disk. In this case, we do not modify the segment head information of that disk segment in order to avoid the high overhead of disk accesses. We only modify the corresponding entry in the segment table: decrease valid_slot_num of that disk segment by 1, set the corresponding slot bit to 0 to indicate that the data block in this slot is invalid now. Then the new data block is added to the LRU cache, the rch_state is set to RCH_LRU_CACHE again. Other fields such as seg_id and slot_id are reset accordingly. The corresponding rapid cache head is added to the heads of both the hash list in data lookup table and rapid LRU list which are the same operations as a). Also, the LRU cache table is modified to reflect this change. The whole process is shown in Figure 10(c).

If we cannot find a rapid cache head in the data lookup table matching this LBA, we know it is a newly coming data block. Then it must be added into the backup cache. First, an unused rapid cache head is taken from Free Rapid Cache Head List, and fields in this rapid cache head are filled according to the information in the buffer head. Second, RAPID-Cache adds this block into the LRU Cache:

1. If the LRU cache still has free space, the content of the block is copied into the LRU cache and its address in the cache is saved in seg_id and slot_id. Its rapid cache head is added to the heads of both the hash list in the data lookup table and the rapid LRU list. The LRU cache table is modified accordingly.

2. In case that the LRU cache is full, RAPID-Cache picks an empty segment buffer, sets it the “current” segment buffer and copies contents of the last 31 data blocks in the rapid LRU list from the LRU cache to this segment buffer. The entries in
both LRU cache table and segment table are modified. Since segment buffers are also located in NVRAM, space used by those blocks in LRU cache are safely freed, and the newly coming data block is inserted into the LRU cache. Also its rapid cache head is added to the heads of both the hash list in the data lookup table and the rapid LRU list. And its entry in the LRU cache table is set. Now RAPID-Cache schedules one large write to copy the entire contents of the current segment buffer to an empty disk segment on the cache disk. RAPID-Cache switches to another empty segment buffer as current segment buffer. Since the entire contents is written to cache disk in one large write instead of many small writes, the segment buffer is available again quickly after the write finishes.

When a block in the primary cache is destaged to the data disk, the corresponding block in the backup cache needs to be invalidated. If this block is in the LRU Cache, the slot occupied by this block is freed and the entry in LRU cache table is set to NULL. If it is in a segment buffer, the valid_slot_num is decreased by 1 and the corresponding entry in the slot_map is marked as invalid; if it is on the cache disk, RAPID-Cache will not modify the corresponding disk segment head on the cache disk (otherwise the overhead will be too high), but simply modifies its in-memory copy in the segment table. In all cases, the rapid cache head of this block is freed and added to the Free Rapid Cache Head List.

Figure 11: Write Operations in RAPID backup cache (Cache Miss)

In summary, in write operations, the backup cache (the small NVRAM together with the cache disk) functions like a large NVRAM write cache and can achieve similar performance and better reliability.

4.2 Reads

The backup cache does not participate in normal read operations. Just like in a regular file system, the Linux system simply searches the requested block in the primary buffer cache, and gets the data from the cache on a cache hit, or fetches the data from the data disk on a cache miss.

4.3 Garbage Collection

If the percentage of used segments on the cache disk reaches a threshold, for example, 70%, RAPID-Cache will start a garbage collection operation. First, the segment table is searched and disk segments which have few valid blocks (fragmented segments) are selected. Second, using ptr_table in the segment table, the valid data blocks in these fragmented segments are copied from the primary cache to a segment buffer. Third, when this segment buffer is full, a large disk write is scheduled to destage its contents to the cache disk. After write finishes, the data lookup table and the segment table are modified accordingly. The fragmented disk segments are freed and can be reused.

As discussed before, in real systems, garbage collection almost never happens, so it basically has no performance impact. We use a 512 MB cache disk in our experiments and found most segments in cache disk are made empty (its valid_slot_num in the segment table reaches 0) quickly. It hardly happens that 70% of the cache disk are used, so there’s no need to perform garbage collection at all.

4.4 Crash Recovery

Crash recovery in RAPID-Cache can be divided into two categories:

4.4.1 System Crash

If the entire system crashes, the modified data can be safely recovered during system reboot time. We set a crash_flag in the system boot sector. During system booting time, RAPID-Cache sets this flag. If the system shutdown correctly, this flag will be cleared. When the system starts, RAPID-Cache checks this flag and starts the crash recovery process if the flag is set. The process is strait-forward. Before any new disk I/O is issued to the data disk, the LRU cache table and segment table are searched, all valid data in the backup LRU Cache, segment buffers and the cache disk are copied back to the data disk. Then all components in the RAPID backup cache are initiated.

4.4.2 Cache Crash

In case that the primary buffer cache is corrupted, each dirty block in it have a copy in backup cache and can be recovered.

In case that RAPID backup cache has problems, for example, the NVRAM or cache disk fails, we can perform a regular system shutdown, which cause data in the primary cache be safely written to the data disk. We then replace the failed component and reboot. If the system supports hot-swappable disks/NVRAM, we can replace the failed device while system keeps running. Then all dirty data blocks in the primary cache are copied to the backup cache.
5 Performance Evaluation

5.1 Benchmarks and Experiment Environment

We created a set of benchmarks similar to the ones used by Nightingale et al. in [18]. The benchmarks consist of three sets of programs: N-user untar, N-user copy and N-user delete. N-user untar simulates one or more users concurrently and independently decompressing and untaring a tar.gz file to create a directory tree. N-user copy and N-user delete simulate one or more users concurrently copying or deleting the entire directory tree created by N-user untar, respectively. We also choose the PostMark disk I/O benchmark [19] to test disk I/O bandwidth. These benchmarks represent typical and realistic I/O intensive tasks performed by normal users on a daily basis.

The particular .tar.gz files we used are the source code distribution tar files of emacs-20.7 and Linux kernel 2.4.12. Emacs is one of the most popular editors in the UNIX environment. The resulting source file tree contains 33 sub-directories and 2051 files with an average file size of 27.45 KB. The total size of the directory tree is 56.3 MB. For Linux kernel 2.4.12, the resulting source file tree contains 501 sub-directories and 9679 files with an average file size of 11.87 KB. The total size of the directory is 112.2 MB.

All experiments were conducted on an Intel Pentium 4 1.8 GHz PC, running Redhat Linux 7.2 with an upgraded 2.4.12 kernel. Standard EXT2 file system is used. The machine has 256 MB memory, two IBM Deskstar 20 GB 7200RPM Ultra ATA/100 drives with an average seek time of 8.5 ms. An MM5415CN NVRAM PCI card from Micro Memory Inc. was used. All the Linux file systems are located on the first hard disk.

We ran benchmarks on the following different system configurations:

1. Baseline (Sync): We used Redhat Linux 7.2 with an unmodified kernel 2.4.12 as a baseline system. In this configuration, Linux file systems are mounted in the synchronous mode. Since file data are cached in the buffer cache, it will cause data loss in case of power failures or system crashes.

2. Baseline (Async): This configuration is the same as Baseline (Sync) except that all Linux file systems are mounted in the asynchronous mode. It has better performance than Baseline (Sync). However, since meta-data updates are asynchronous, it does not guarantee file system consistence after crash, and is not suitable for most applications.

3. NVRAM: We used NVRAM as a regular second level buffer cache. The NVRAM sizes used are 4 MB, 16 MB, 64 MB and 128 MB. Each time when a write request comes in, no matter it is a synchronous or asynchronous write, we write it to the NVRAM. A write complete signal is then sent back to the system. If the NVRAM is full, we choose several LRU dirty blocks and write them back to disk using synchronous writes. This NVRAM functions as a big non-volatile buffer cache. In this configuration, all file systems are mounted using synchronous mode.

4. RAPID: The backup cache uses 3.25 MB of NVRAM (2 MB for the LRU Cache, 1 MB for Segment Table, and two 128 KB segment buffers) unless otherwise noted. The cache disk is a 512 MB disk partition on the second hard disk with no Linux file system partitions on it. In this configuration, all file systems are mounted using synchronous mode.

We reboot the system before executing a experiment every time, thus the benchmarks are running with a cold cache effect.

5.2 Execution Time

![Figure 12: Benchmark Execution Time for 1-user untar, copy and delete of Emacs](image)

Figure 12: Benchmark Execution Time for 1-user untar, copy and delete of Emacs

![Figure 13: Benchmark Execution Time for 1-user untar, copy and delete of Linux Kernel](image)

Figure 13: Benchmark Execution Time for 1-user untar, copy and delete of Linux Kernel

We measured the performance of these configurations in terms of execution time. Both 1-user and 4-user operations are tested, the results are shown in Figures 12, 13 and 14. We used the Unix time command to measure benchmark execution times, the user time and system CPU time. All numerical results shown here are the average of three runs.

For the two baseline systems, the results show that Baseline (Sync) has the worst performance in all cases, because of the high cost of synchronous meta-data updates. Baseline (Async) in general has the best performance (except that in 1-user untar of Linux). However it does not guarantee file system consistence therefore is not suitable for most applications.

Using NVRAM as write caches can improve both performance and reliability. When the cache size is small (4 MB),...
NVRAM (4MB) performs slightly better than Baseline (Sync), and the reliability is improved also since dirty data are cached in NVRAM instead of RAM. As the cache sizes increases to up to 120 MB, the performance gain noticeably improves also.

RAPID-Cache has the best performance/reliability combination among all systems. Only Baseline (Async) is slightly faster than RAPID (3.25MB), because the latter has to perform much more work (keeping two copies of data) in order to provide much better reliability. In fact, with only 3.25 MB of NVRAM, RAPID (3.25MB) outperforms NVRAM (120MB) in all but three cases. And the former always performs much better than NVRAM (64MB).

In addition to high performance, RAPID-Cache also has better reliability than others. It is certainly much more reliable than Baseline (Sync) and Baseline (Async), since both dirty data and meta-data have non-volatile copies in the backup caches. RAPID-Cache is also more reliable than NVRAM caches because (1) RAPID Cache always keeps two copies of data; and (2) most data blocks in the backup cache are stored in the cache disk, which has a higher MTTF than NVRAM. Detailed analyses on reliability on RAPID-Caches and NVRAM caches can be found in [10, 11].

5.3 LRU Size Effects

All previous results of RAPID-Caches were obtained with 3.25MB of NVRAM (The LRU cache size is 2 MB). To evaluate the influence of LRU cache size on system performance, we re-executed 1-User operations in RAPID-Cache implementation with changed LRU cache sizes from 256 KB to 8192 KB. The results are shown in Figures 15, 16 and 17.

Increasing the LRU cache size from 256 KB to 1 MB can significantly improve performance, as larger caches can exploit disk request locality more effectively. However, beyond 1 MB, further increasing the cache size does not results in noticeable performance gain. It is worth to compare such a result with those of NVRAM caches in Figures 12, 13 and 14, where increasing the NVRAM size from 64 MB to 128 MB can still significantly improve performance. This shows that RAPID-Cache can very efficiently utilize the NVRAM. A small NVRAM in front of a cache disk creates a large, virtual NVRAM.

5.4 Bandwidth

The last experiment was to test the system bandwidth, using the PostMark benchmark. We simulated creating 10000 files in 10 directories with the file size ranging from 512 bytes to 16384 bytes, doing some read/write operations and deleting these directories finally. For each configuration, we repeat the experiment 10 times. The results of the average bandwidth are show in Figure 18.

Baseline (Sync) has the lowest bandwidth. Which is not a surprise. For NVRAM caches, as the cache size increases, their bandwidth improves also. NVRAM (120MB) is more than 12 times faster than Baseline (Sync). With only 3.25 MB of NVRAM, the RAPID-Cache achieve a bandwidth between NVRAM (64MB) and NVRAM (128MB), and is more than 10 times faster than Baseline (Sync).

6 Related Work

Ng and Chen [20, 21] proposed Rio (RAM I/O) to implement reliable RAM. Rio is a software system that makes RAM to survive operating system crashes. The goals of RAPID-Caches and RIOs are very different. RAPID-Caches are hardware solutions that tolerate hardware failures, power outages and software crashes. Rio focus on surviving software crashes only.
7 Conclusion

We present the design and implementation of RAPID-Cache for the Linux operating system. We conduct measured experiments to confirm the simulation results in [10, 11]. Four kinds of benchmarks are used on seven different configurations.

The measurement results show that RAPID-Cache is an inexpensive way to make the buffer cache non-volatile, highly reliable and high performance. Using only 3.25 MB of NVRAM, a RAPID-Cache outperforms conventional NVRAM caches of 64 MB or even 128 MB. RAPID-Cache also provide better reliability than conventional NVRAM caches because of data redundancy, and because disks have a higher MTTF than NVRAMs.

Finally, unlike other log-based storage techniques such as LFS or DCD, where garbage collection or destaging cost may become a problem, RAPID-Cache virtually has no garbage collection overhead.

We plan to make the source code and binary code of RAPID-Cache available to the public as soon as possible.

Acknowledgments

This work is supported in part by the National Science Foundation Career Award CCR-9984852, and an Ohio Board of Regents Computer Science Collaboration Grant. Sudhindra Rao participates in this research.

References


