On Design and Implementation of a Large Virtual NVRAM Cache for Software RAID

Xubin He — Qing Yang

Department of Electrical and Computer Engineering
University of Rhode Island, Kingston, RI 02881
U.S.A

hexb@ele.uri.edu
qyang@ele.uri.edu

ABSTRACT: *In order to overcome the small write problem in RAID5, especially software RAID5, we have designed and implemented a software RAID with a large virtual NVRAM cache under the Linux kernel. Because no additional hardware is needed to implement our write cache, we named it Virtual NVRAM Cache or VC-RAID for short. The main idea is to use a combination of a small portion of the system RAM and a log disk to form a hierarchical cache. The log disk can be either a dedicated physical disk or several partitions of disks in the RAID. Since the log disk quickly absorbs write data from the small RAM, this hierarchical cache appears to the host as a large nonvolatile RAM. A prototype VC-RAID implemented under the Linux kernel has been tested for an extended period of time to show that it functions properly. Performance measurements have been carried out using typical programs and popular benchmarks such as ServerBench 4.1, PostMark and Bonnie. Our measurements show that the VC-RAID has superb performance advantages over the built-in software RAID shipped with the Linux package. Depending on the workload characteristics, performance gains due to VC-RAID range from 67.7% to an order of magnitude. For applications that have data locality, we observed up to a factor of 16 performance improvements in terms of user response time. In many situations, VC-RAID achieves similar performance as RAID0 and some time better than RAID0 indicating that VC-RAID realizes the maximum potential to hide small write problems in RAID5.*

KEY WORDS: *Parallel I/O, RAID, cache, VC-RAID, small write*

1. Introduction

Software RAID has become very popular for applications that require reliable and economic storage solutions as E-commerce emerges. Most commodity operating systems such as Windows, Solaris, and Linux have built-in software RAIDs. While these built-in software RAIDs provide immediate performance benefits at lowest cost, most software RAID installations shy away from RAID5 configuration, the most popular RAID configuration in storage industry, and therefore lose the benefit of high reliability of RAID. The main reason why RAID5 is not installed in software RAID is performance penalty resulting from small write operations. Each such a small write requires 4 disk operations: reading old data, reading old parity, writing new data, and writing new parity. As a result, for workloads consisting of a mix of read and write operations, software RAID5 shows very poor performance and becomes impractical.

The most popular and practical solution to small write problems is to use large write cache. Modern RAID systems make extensive use of nonvolatile RAM (NVRAM) write caches to allow fast write [CHE 94][HOU 97][MEN 93][TRE 95], or asynchronous write. Write requests are acknowledged before they go to disk. Such write caches significantly reduce user response times of RAID. Large write caches can also improve system throughput by taking advantages of both temporal locality and spatial locality of general workloads. Treiber and Menon reported that write caches could reduce disk utilization for write by an order of magnitude when compared to standard RAID5 [TRE 95]. In fact, some researchers have argued that the use of large caches in RAID systems has rendered debates over the best RAID level irrelevant [COO 96].

While most commercial hardware RAID systems have large write caches for better performance, there is no readily applicable cache solutions for software RAID or do-it-yourself RAID [ASA 95]. This is because that write cache has to be nonvolatile to be reliable. Adding a nonvolatile RAM into software RAID not only is costly but also requires special hardware devices to interface to the disks. Our objective here is to present a simple software cache solution that can be embedded into the software RAID package shipped with a commodity OS with no additional hardware.

Our approach is to use a combination of a small RAM and a log disk to form a large and nonvolatile write cache. The small RAM (a few megabytes) is obtained from a raw partition of the system RAM. The log disk can reside on one physical disk or on partitions of several disks in the RAID. The combination of the RAM and the log disk form a hierarchical write cache. Disk writes are first collected in the small RAM to form a log that is quickly moved to log disk sequentially so that the RAM is emptied for further writes. It appears to the host that there is a very large RAM cache since there is always space for new writes. At the same time, such a large “RAM” has a very high reliability because the small RAM is used only for collecting log and data stay in the RAM for a very short period of time before moving to cache disk that is nonvolatile and more reliable. This large virtual RAM cache hides the small write penalty of RAID5 by buffering small writes and destaging data back to RAID with parity computation when disk activity is low (We are using a thread to monitor the system kernel activity dynamically similar to the command “*top*” in Linux). This approach is different from the write-ahead logging [GRA 93][MOH 95] in that we use a cache disk which is nonvolatile and inexpensive, and also we don’t need to build a new file system as in [CHU 92][MAT 97][SEL 93][SHI 95].

We have designed and implemented this virtual RAM cache under the Linux operating system. Extensive testing of the implementation has been carried out to show it functions properly. We have also carried out performance measurements and comparisons with existing software RAID using typical disk I/O bound programs and popular benchmarks such as ServerBench, PostMark and Bonnie. Our measurements on ServerBench show that the RAID5 with our virtual RAM cache has superb performance advantages over the built-in software RAID shipped with the Linux package. Up to a factor of 3 performance improvements have been observed in terms of transactions per second. Measurements on Bonnie also show similar performance improvements. When we measure the performance of realistic application programs under Linux, we observed much greater performance gains because realistic applications show data locality. An order of magnitude performance improvement was observed in terms of user response time. In many situations, *VC-RAID* achieves similar performance as RAID0 and some time better than RAID0 indicating that *VC-RAID* realizes the maximum potential to hide small write problems in RAID5.

The paper is organized as follows. In the next section, we present detailed concept and design of the large virtual NVRAM cache. Section 3 presents the implementation details. Benchmark and performance results are presented in Section 4. We discuss previous related research work in Section 5 and conclude our paper in Section 6.

2. Architecture and Design of the Virtual NVRAM Cache

In the absence of physical NVRAM cache in software Do-it-yourself RAID, we try to make use of existing system resources and implement a virtual NVRAM cache by means of software. Our idea is very simple. We use a small raw partition of the system RAM and partitions of disks in the RAID to form a cache hierarchy. A thin layer of driver program is inserted between the Linux kernel and the physical device driver for disks. This driver program manages the operation of the cache hierarchy in the way described below.

The general structure of the virtual NVRAM cache has a two level hierarchy: a small RAM buffer on top of a disk called *cache disk*. Small write requests are first collected in the small RAM buffer. When the RAM buffer becomes full or a programmable timer is out, all the data blocks in the RAM buffer are written to the cache disk in several *large data transfers*. Each large transfer finishes quickly since it requires only one seek instead of tens of seeks. As a result, the RAM buffer is very quickly made available again to accept new incoming requests. The two-level cache appears to the host as a large virtual NVRAM cache with a size close to the size of the cache disk. When the system I/O activity is low, it performs *destaging* operations which computes parity and transfer data from the cache disk to the disk array, referred to as *data disks* or *data disk array*. The destaging overhead is quite low, because most data in the cache disk are short-lived and are quickly overwritten therefore requiring no destaging at all [HU 96]. Moreover, many systems, such as those used in office/engineering environments, have shown significant temporal and spatial data locality giving rise to sufficient long idle periods between bursts of requests. Destaging can be performed in the idle periods therefore will not interfere with normal user operations at all. Since the cache disk is a disk with a capacity much larger than a normal NVRAM cache, it can achieve very high performance with much less cost. It is also nonvolatile thus highly reliable.

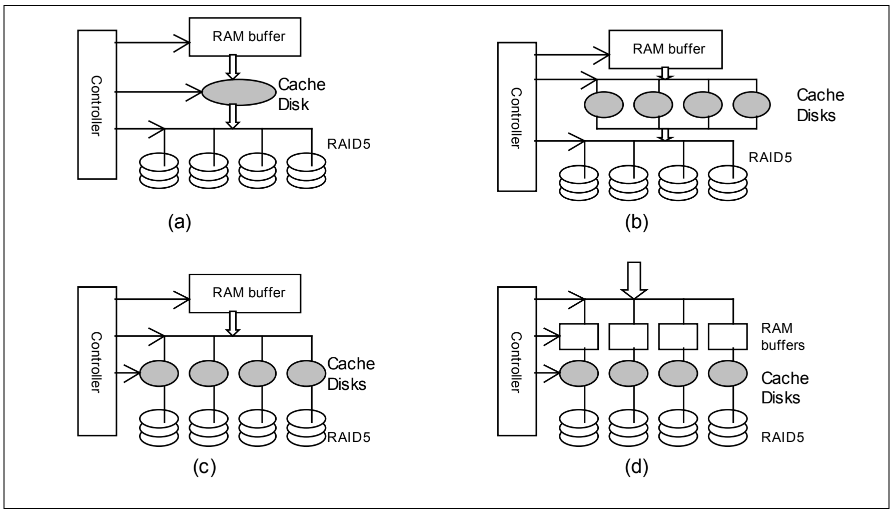

Figure 1. *Possible approaches to VC-RAID. (a) one RAM buffer and one cache disk (b) one RAM buffer and several cache disks (c) one RAM buffer and several cache disks, each cache disk is associated with a disk in the array (d) Several RAM buffers and cache disks, each RAM buffer and cache disk are associated with a disk in the array.*

The cache disk in the virtual NVRAM cache can be a separate physical disk dedicated for caching purpose. Given the low cost of today's hard drives, it is quite feasible to have one of the disks in the disk array to carry out the cache disk function. The cache disk can also be implemented using a collection of logical disk partitions from disks in the RAID with no dedicated cache disk. Depending on how the cache disk is formed, we have four alternative architecture configurations for *VC-RAID* as illustrated in Figure 1.

In the first configuration, a RAM buffer and a physical cache disk are used to cache data to be written to the entire disk array (Figure 1a). This configuration is referred to as *Single Cache Disk* (*scd*) configuration. All data to be written to the array are first written to the RAM buffer. When the RAM buffer is full or the cache disk is idle, the data in the RAM buffer are transferred to the cache disk by a kernel thread, called *RAIDDestage* thread. The *RAIDDestage* thread combines small write requests writes into large one and write to the cache disk at a time. The data in the cache disk are destaged to the data disk array during the system idle time and/or when the cache disk is full. We can choose the size of the cache disk to be large enough to ensure that most of the destages occur during the system idle time.

The above configuration (*scd*) is effective for a small array consisting of a few disks. With the increase of the number of disks in the array, single cache disk in the above configuration may become a new system bottleneck since all write operations and some read operations are performed at this cache disk. To avoid this bottleneck problem, the second configuration uses *Multiple Cache Disks (mcd)* as shown in Figure 1b. Several cache disks collectively form the cache disk in the virtual NVRAM hierarchy. These cache disks are logical partitions physically residing on the data disk array. All these logical partitions form a uniform disk space to cache all write data to the data disk array. When the RAM buffer is full or there are idles cache disks, the destage thread is invoked to move data from RAM buffer to one of the cache disks. A round-robin algorithm is used to determine which cache disk is used next in a log write. Writing to and reading from cache disks can therefore be done in parallel reducing the bottleneck problem.

The above two configurations (*scd* and *mcd*) both have a global cache that cache data for the entire disk array. It is also possible to have private write caches for each disk in the disk array. That is, there is a virtual NVRAM cache for each individual disk in the disk array. Figure 1c shows this configuration where a cache disk is associated with each data disk in the array and only cache data for the particular data disk below it. If the RAM buffer is also managed individually for each data disk, then we have the fourth configuration as shown in Figure 1d.

3. Implementation

This section presents data structures and algorithms used in implementing the *VC-RAID*. We have implemented the *VC-RAID* as a loadable module under Linux kernel 2.0.36 based on the *MD* (*Multi-Device*) driver. The *md Multi-Device* kernel module, included as a standard part of the v2.0.x kernels, provides RAID-0 (disk striping) and multi-disk linear-append spanning support in the Linux kernel. The RAID 1,4 and 5 kernel modules are a standard part of the latest 2.1.x kernels; patches are available for the 2.0.x kernels and the earlier 2.1.x kernels.

Let us first look at how disk requests are processed on Linux. Take a synchronous write as an example. As shown in Figure 2a, a user program issues a write request by calling *fwrite().* The system switches to the kernel mode through the system call *write().* Then the Linux generic block driver accepts the requests through *ll_rw_block()* and issues a request to the low-level physical disks through *make_request().* The physical disk driver finishes the request by *hd_request()* (IDE disks) or *sd_rquest* (SCSI disks). Figure 2b shows the case of software RAID, where *md* driver accepts the requests and determines which disk in the array services the request and computes the parity, finally writes the data and parity to the corresponding disks through low level physical disk drivers. Figure 2c shows how *VC-RAID* processes a write request. *VC-RAID_request* accepts all requests to the array, and combines small writes into large ones in the RAM buffer firstly. Whenever the cache disk is idle, the data in the RAM buffer are moved to the cache disk in a large write. During the system idle time, data in the cache disk will be destaged to disk arrays.

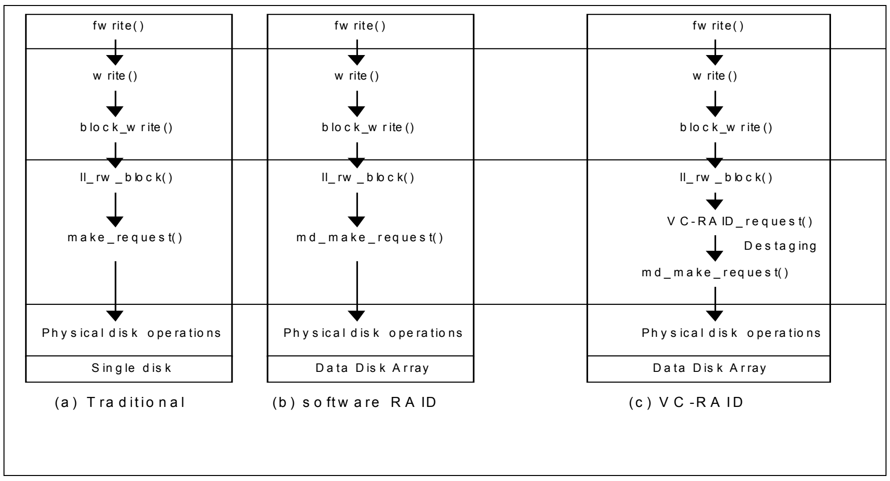

Figure 2. *Procession of write requests.*

3.1. *In-memory Data Structure*

During the system initialization, a fixed amount of physical RAM is reserved for our *VC-RAID*, all the in-memory data structures reside in this RAM area. The In-memory data structure includes a Hash table which is used to locate data in the cache, a data buffer which contains several data slots and a number of In-memory headers (Figure 3). The In-memory headers form two lists: the Free List which is used to keep track of free slots and the LRU list which traces the Least Recently Used slots.

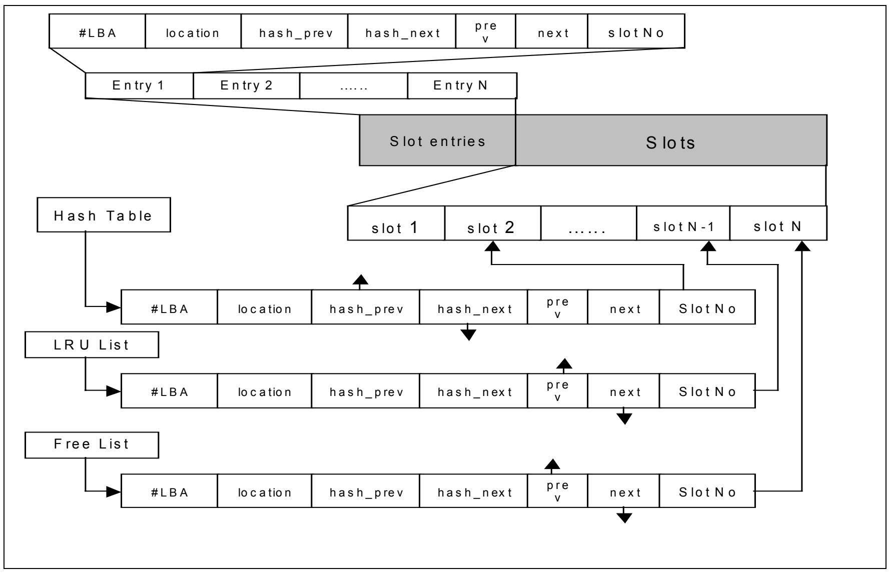

Figure 3. *RAM buffer layout. RAM buffer consists of slot entries and slots. The hash table, LRU list and Free list are used to organize the slot entries.*

In our implementation, a data block may exist in one of the following places: the RAM buffer, the cache disk or the data disks. A data Lookup Table is used to keep track of the data location in these places. Since the driver must search the table for every incoming request, it is imperative to make the searching algorithm efficient. In Linux kernel 2.0.36, the default file system is *ext2* file system. When a file system is created by *mke2fs*, the block size, *b_size,* will be fixed. The default is 1024 byte per block, or it can be specified to be 1024, 2048 or 4096. After a file system is created, the LBAs (*Logical Block Address*) of all requests are aligned to the *b_size* boundary. And all atomic request size equals to *b_size*. We use a hash table to implement the data lookup table with the LBA of incoming requests being the search key, and the slot size being the size of a block, as shown in Figure 3. A slot entry consists of the following fields:

- ❑ An *LBA* entry that is the LBA of the cache line and serves as the search key of hash table;
- ❑ A *location* field is divided into three parts:

1) A cache disk index (8 bits), used to identify a cache disk in multiple cache disk configuration. It can support up to 256 cache disks. In single cache disk configuration, this field is set to zero;
2) A state tag (4 bits), used to specify where the slot data is: IN_RAM_BUFFER, IN_CACHE_DISK, IN_DATA_DISK or SLOT_FREE;
3) A cache disk block index (20 bits), used to specify the cache disk block number if the state tag indicates IN_CACHE_DISK. The size of each cache disk can be up to 1048576 blocks.

- ❑ Two pointers (*hash_prev* and *hash_next*) are used to link the hash table;
- ❑ Two pointers (*prev* and *next*) are used to link the LRU list and FREE list;
- ❑ A *Slot-No* is used to describe the in-memory location of the cached data.

3.2. *Cache Disk Organization*

The organization of cache disk is much simpler compared to that of RAM buffer. The cache disk consists of cache disk headers and an amount of physically consecutive blocks. The block size equals to the slot size in RAM buffer. A cache disk header comprises a number of cache-disk-block-index/array-LBA pairs which are used to track the locations of cached data blocks in the data disk array. The cache disk headers are only for crash-recovery purpose and are never accessed during normal operations.

3.3. *Basic Operations*

3.3.1. Write

After receiving a write request, the *VC-RAID* driver first searches the Hash Table. If an entry is found, the entry is overwritten by the incoming write. Otherwise, a free slot entry is allocated from the Free List, and the data are then copied into the corresponding slot, and its address is recorded in the Hash table. The LRU list and Free List are updated. The driver then signals the kernel that the request is complete, even though the data has not been written into the disk array. This immediate report scheme does not cause any reliability problem as will be discussed in section 3.4. If user applications set "*O_SYNC*" flag in the write request, the driver writes the data to the cache disk directly and wait until the requests finishes before signalling the kernel that the request is complete.

3.3.2. Read

After receiving a read request, the *VC-RAID* driver searches the Hash Table to determine the location of the data. In our case, data requested may be in one of three different places: the *VC-RAID* RAM buffer, the cache disk(s), or the data disk array. If the data is found in the RAM buffer, the data are copied from the ram buffer to the requesting buffer, and then the driver signals the kernel that the request is complete. If the data is found in the cache disk(s), the data are read from the cache disk into the requesting buffer; otherwise, the *VC-RAID* driver forwards the read request to the disk array.

3.3.3. Destages

There are two levels of destages: detaging data from the RAM buffer to the cache disk (*Level 1 destage*) and destaging data from cache disk to data disk array (*Level 2 destage*). We implement a separate kernel thread *RAIDDestage* to perform the destaging tasks. The *RAIDDestage* thread is registered during system initialization and monitors the *VC-RAID* states. The thread keeps sleep at most of the time, and is activated when the *VC-RAID* driver detects an idle period or the *VC-RAID* RAM buffer and/or the cache disk becomes full. *Level 1 destage* has higher priority than the *Level 2 destage*. Once the *Level 1 destage* starts, it continues until the RAM buffer becomes empty and it is uninterruptible. Once the *Level 2 destage* starts, it continues until the cache disk becomes empty, or until a new request comes in. In the later case, the destage thread is suspended, until the driver detects another idle period.

As for *Level 1 destage*, the data in the RAM buffer are written to the cache disk sequentially in large size (up to 64K). The cache disk header and the corresponding in-memory slot entries are updated. In the multiple-cache-disk (*mcd*) configuration, a round-robin algorithm is used to select the cache disk to receive data. All data are written to the cache disk(s) in “append” mode, which ensures that every time the data are written to consecutive cache disk blocks.

For *Level 2 destage*, we use a “last-write-first-destage” algorithm according to the LRU List. Each time a data segment (64Kb in our preliminary implementation, and it is a configurable parameter to make use of optimal striping unit size [CHE 95]) are read from the consecutive blocks of the cache disk and then written to the disk array in parallel. At this point, parity code is calculated and written to the parity block. After data is destaged to the data disk array, the LRU list and free list are updated subsequentially.

3.4. Reliability Analysis

One potential reliability problem with this virtual NVRAM cache is that the cache disk fails before data are destaged to the data disk array. The cache disk may become a potential single point of failure. To address this problem, we can mirror the cache disk by using another partition on different disk which acts as a backup cache disk. In this case the *Level 1 destage* writes data from the RAM buffer to cache disk and backup cache disk in parallel. Since the total size of a good performance cache is usually in the range of 1% of total data volume of the storage system, we would not expect significant waste in terms of disk space. On the other hand, the cost of disk is rapidly dropping and it is quite practical to have one percentage point increase in disk space to trade for high reliability.

VC-RAID uses immediate report mode for writes as discussed in section 3.3.1, except for those requests with the "*O_SYNC*" flags. This immediate report does not cause a serious reliability problem for the following reasons: First, the delay caused by the *VC-RAID* driver is limited to several hundreds of milliseconds at most. The driver writes the data to the cache disk within several hundreds of milliseconds. As default, the Linux caches file data and metadata in RAM for 30 and 5 seconds, respectively, before they are flushed into the disks. We believe that the additional several hundreds of milliseconds should not cause any problem. In fact, we can use a tracking structure to track the metadata dependency and adopt a mechanism similar to Soft Updates [MCK 99] to further protect the metadata integrity and consistency. Second, the *VC-RAID* RAM buffer is reserved during the system boot and is not controlled by the OS kernel. It is isolated from the other part of RAM used by the system and used exclusively by *VC-RAID* driver. It should have less chance to be crashed by other applications. Finally, a small amount of NVRAM can be used in the real implementation to further guarantee the reliability before the data is written to the cache disks.

Our current implementation of *VC-RAID* is for RAID level 5. As soon as data are destaged to the data disk array, they are parity protected in the same way as RAID 5.

Another issue about reliability is crash recovery. In case of system crash, data can be recovered from the cache disk. The system first switches to the single user mode. All data that have not destaged to the data disk array are written to data disk array according to the cache disk headers which record the relationships between a cache disk block and the corresponding data disk block. At this point the *VC-RAID* returns to a clean and stable state. Then the file system can start the normal crash-recovery process by running "*e2fsck*".

4. Performance Evaluations

4.1. Experimental Setup

We installed and tested our drivers on a Gateway G6-400 machine running Linux 2.0.36. The machine has 64MB DRAM, two IDE interfaces and two sym53c875 SCSI adapters. Six hard disks (including 4 SCSI disks and 2 IDE disks) are connected to this machine. The disk parameters are listed in Table 1. In the following benchmark tests, unless specified otherwise, the size of *VC-RAID* RAM buffer is 4M. A cache disk size is 200MB with data block size being 1KB. The 4

SCSI disks are used to form the disk array while the OS kernel runs on the first IDE disk (model 91366U4). The second IDE disk (model 91020D6) is used as cache disk in the single cache disk (*scd*) configuration. For the case of the multiple cache disk(*mcd*) configuration, two logical partitions, one from each of the IDE disks, are used as cache disks.

Disk Model	Interface	Capacity	Data buffer	RPM	Latency (ms)	Transfer rate (MB/s)	Seek time (ms)
DNES-318350	Ultra SCSI	18.2G	2MB	7200	4.17	12.7-20.2	7.0
DNES-309170	Ultra SCSI	9.1G	2MB	7200	4.17	12.7-20.2	7.0
91366U4	ATA-5	13.6G	2MB	7200	4.18	Up to 33.7	9.0
91020D6	ATA-4	10.2G	256KB	5400	5.56	18.6	9.0
ST52160N	Ultra SCSI	2.1G	128KB	5400	5.56	10	11
ST32151N	SCSI-2	2.1G	256KB	5400	5.54	10	10.4

Table 1. *Disk parameters.*

The performance of *VC-RAID* is compared to that of the standard software RAID0 and RAID5 shipped with the Linux operating system. To ensure that the initial environments are the same for every test, we performed tests for each benchmark run in steps as follows:

1) Load the corresponding driver *VC-RAID*, RAID0 or RAID5 (using "*insmod*");
2) Execute the benchmark program;
3) Unload the driver (using the command "*rmmod*");
4) Reboot the system and prepare for the next test.

4.2. *Benchmarks*

Workload plays a critical role in performance evaluations. We paid special attention in selecting workload for our performance testing. In order to give a realistic performance evaluation and comparison, we use real world benchmarks in our performance measurements. Benchmark programs that are used in our performance measurements are ServerBench which is a very popular benchmark for testing server performance, Bonnie which is a standard benchmark used for unix systems, PostMark which is a popular file system benchmark and some user application programs that are disk I/O bound such as *untar, copy*, and *remove*.

4.2.1. ServerBench 4.1

ServerBench [SER 2000] is a popular benchmark developed by ZD Inc. and has been used by many organizations including ZD Lab to measure the performance of application servers in a client/server environment. It consists of three main parts: a client program, a controller program and a server program. A server, a controller and several clients are needed to run server program, controller program and client program respectively. The aforementioned G6-400 machine act as the server in the test, and 5 PCs with Celeron 500MHZ CPU, 64MB DRAM acts as controller and clients which run windows 98. All machines are equipped with 3C905B 10/100M network adaptors and interconnected through a D-Link 10/100M 8 port switch. Our test suites are based on the standard test suite provided by ServerBench. The size of data set is 128 MB, which is divided into 4 data segments with each segment being 32MB.

The performance metric used in this measurement is *Transactions Per Second*(TPS) which is the standard performance measure as output of ServerBench. For each run, we measured the TPS results of standard software RAID0, RAID5, *scd* (Single Cache Disk) *VC-RAID*, and *mcd* (Multiple Cache Disk) *VC-RAID* under the same conditions. Within each run, there are 4 test sets corresponding to 1 to 4 clients respectively. For each test set, there are 16 groups (mixes), each with 32 transactions. Every transaction comprises 3 basic steps: client requests (read or write) to the server; server accepts and processes the request; and finally client receives a response from the server. The percentage of write requests, referred to as *write ratio*, is an important parameter to be determined in each test round. We determine the write ratio based on our observations of existing disk I/O traces and previously published data.

4.2.2. Bonnie

Bonnie [BON 96] is a benchmark used to measure the performance of Unix file system operations. It performs a series of tests on a file of known size. For each test, Bonnie reports the results of sequential output, sequential input and random seek time for both block device and character devices. Since disk driver and RAID driver are both block devices, in our experiments we are only concerned with performance of block devices. We measured write performance and random seek performance for block devices. For block writes, a file is created using system call *write()*. For random seeks, we run 4 concurrent processes operating on one large file of 100MB. The 4 processes perform a total of 200,000 *lseek()s* to random locations computed using *random()*. Each such random seek is followed by a read operation using system call, *read()*. Among all these random read operations, 10% of them are rewritten immediately in the same location.

4.2.3. Postmark

PostMark [KAT 99] is a popular file system benchmark developed by Network Appliance. It measures system throughput in terms of transaction rates in an ephemeral small-file environment by creating a large pool of continually changing files. "PostMark was created to simulate heavy small-file system loads with a minimal amount of software and configuration effort and to provide complete reproducibility [KAT 99]." PostMark generates an initial pool of random text files ranging in size from a configurable low bound to a configurable high bound. This file pool is of configurable size and can be located on any accessible file system. Once the pool has been created, a specified number of transactions occur. Each transaction consists of a pair of smaller transactions, i.e. *Create file or Delete file* and *Read file or Append file.* Each transaction type and its affected files are chosen randomly. The read and write block size can be tuned. On completion of each run, a report is generated showing some metrics such as elapsed time, transaction rate, total number of files created and so on.

4.2.4. Untar/Copy/Remove

We have created a set of our own metadata intensive benchmark programs. This benchmark consists of 3 sets of script programs: *untar, copy*, and *remove. Untar* creates a directory tree by untaring a tar file (using "*tar xf*" command). The particular tar file we used is the standard test suites file of web-bench by ZDNet. To make the data set larger, we double the test suites. The resulting file tree contains 952 subdirectories with different depths and 12322 files with an average file size being 9.9 KB. The total size of the directory tree is 122MB. This source directory tree resides in a local system disk. The *copy* and *remove* perform users' copying (using the "*cp –r*" command) and removing (using the "*rm –rf*" command) the entire directory tree created by the *untar* test. The performance metric used for this benchmark is user response time since in this case user would be mostly interested in response time. The smaller number means better performance.

4.3. Numerical Results and Discussions

With the experimental settings and the benchmarks described above, we have carried out extensive experiments to evaluate the performance of *VC-RAID* and compare it with standard software RAID0 and RAID5 shipped with Linux package. All numerical results presented in this subsection are the average of 3 experimental runs. In our tests, ***mcd*** and ***scd*** stand for VC-RAID with multiple cache disks and VC-RAID with single cache disk, respectively.

4.3.1. Transactions Per Second (TPS)

Figure 4 shows the average transactions per second (TPS), the performance score output by ServerBench, as a function of number of clients. Three separate figures are plotted corresponding to different write ratios. As mentioned previously, the write ratios were selected based on our observation of typical disk I/O traces. For example, our observation of HP traces (Snake, Cello, and hplajw) [RUE 93] and EMC traces [HU 99] indicate that average write-ratio in a typical office and engineering environment is about 57%. Hua et al [HUA 99] used write ratios of 0.1, 0.5 and 0.9 in their performance evaluation for cached RAID system. From Figure 4a, we can see that when the write ratio is very low (0.1) the performances of *VC-*

RAID, standard software RAID0 and RAID5 are very close. One can hardly notice the difference in terms of TPS for these different RAID systems implying that the software RAID performs fine with such low write ratio.

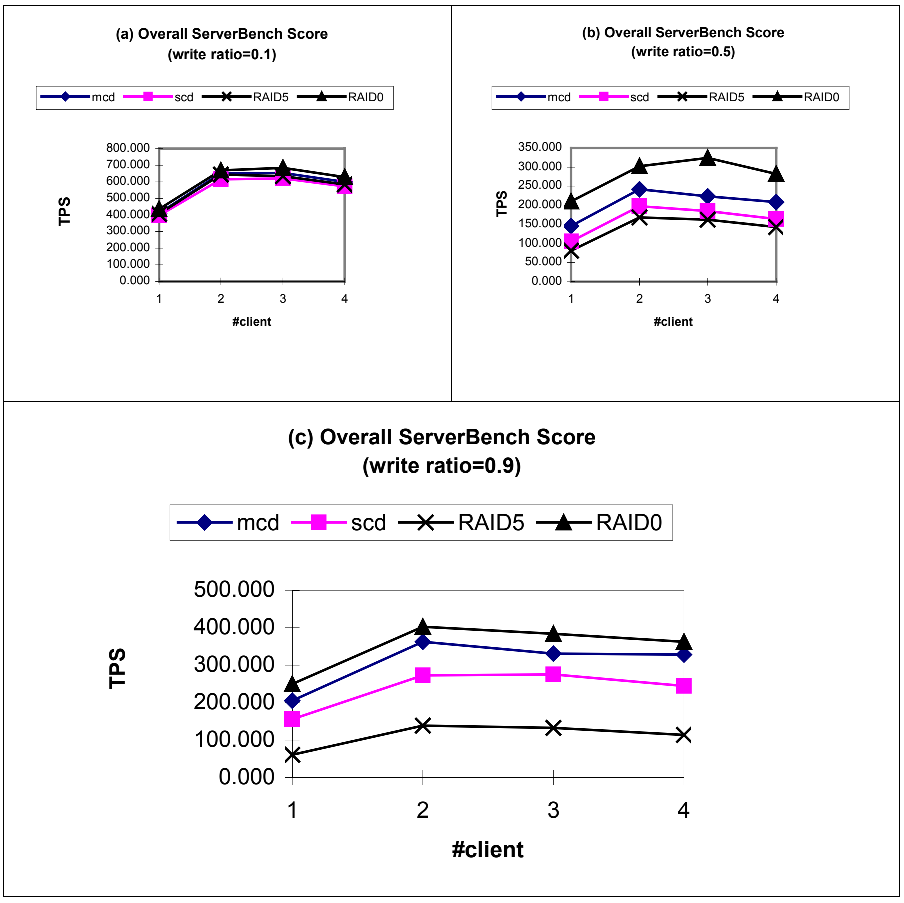

Figure 4. *VC-RAID vs. RAID5 and RAID 0 (Mean request size=1k).*

As the system RAM size increases in today's server computers, a large portion of disk read operations are cached in the system RAM. That is, the large system RAM filtered out many disk read operations. As a result, the proportion of write operations as seen by the disk I/O system increases. When the write ratio increases, the performance penalty due to small write problems of RAID5 comes into picture. Figures 4b and 4c show the TPS for write ratios being 0.5 and 0.9 respectively. For these types of workloads, our *VC-RAID* shows significant performance advantages. With multiple cache disks (*mcd*), the TPS number is doubled compared to standard software RAID5 for write ratio of 0.5 and more than tripled for write ratio of 0.9.

We noticed in all our measurements that TPS drops after the number of clients exceeds 2. This result can be mainly attributed to the network congestion caused by NIC at the server. The 3C905B 10/100M network adapter at the server is not fast enough to handle large number of requests from more clients.

In order to observe how much the *VC-RAID* can hide the small write problem of RAID5, we also compared the performance of *VC-RAID* to that of software RAID0. Since RAID0 has no redundancy, there is no overhead for write operations such as parity computations and parity updates. Figures 4b and 4c show that *mcd* can realize about 80% performance of RAID0 for write ratio of 0.5 and over 90% performance of RAID0 for write ratio of 0.9. The reason for the performance gap between the *VC-RAID* and RAID0 can be explained as follows. The workload to the server generated by ServerBench is fairly random with no spatial and temporal localities. As a result, destage operations may affect the overall performance because the system cannot find idle period to do destage operations. Further more, some read operations may find data in cache disk in the *VC-RAID* limiting the parallelism for these reads. This is why the performance gap is getting smaller as write ratio increases as shown in Figure 4c.

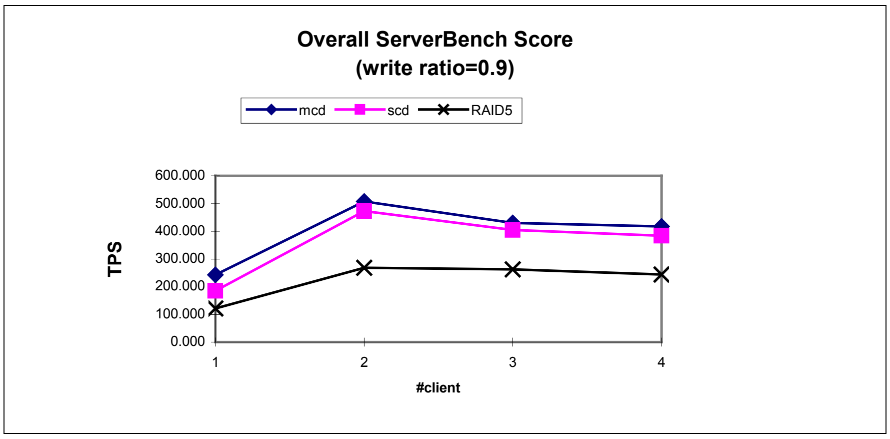

Figure 5. *scd vs. mcd (mean request size=0.5k).*

We noticed performance different between single cache disk (*scd*) *VC-RAID* and multiple cache disk (*mcd*) *VC-RAID* in Figure 4b and 4c. We believe that this difference mainly results from destage overhead. In order to verify this, we change the average request size issued from clients to the server from 1 KB to 0.5 KB. The measured results are shown in Figure 5. As is shown, with smaller request size, the performance difference between *scd* and *mcd* becomes smaller. This is because the frequency of destage operation is smaller due to small data sizes. For large request sizes, multiple cache disk allows destage operations to be performed in parallel resulting in better performance than single cache disk case.

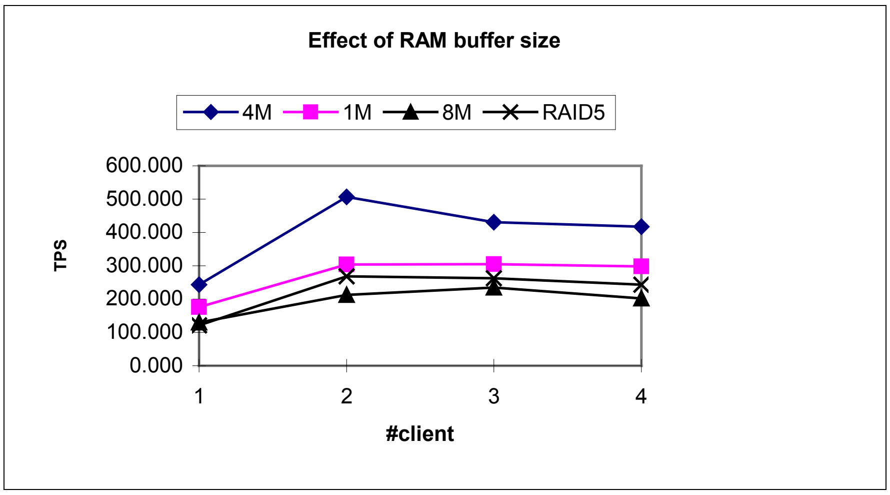

Figure 6. *Effect of RAM buffer size.*

To examine the effect of the size of the RAM buffer reserved for *VC-RAID* on the overall performance of the *VC-RAID* system we plotted the TPS as a function of RAM buffer size as shown in Figure 6. Our measurement results show that 4 MB RAM buffer gives optimal performance. Smaller RAM may get filled up quicker than cache disk can absorb resulting in wait time for some I/O requests. If the RAM buffer is too large, on the other hand, the remaining system RAM used for regular file system becomes small adversely affecting the overall performance. Depending on the workload environment, the optimal RAM buffer size should be tuned at set up time.

4.3.2. Response Times

Our next experiment is to measure and evaluate user response times by running realistic application programs under Linux operating system such as *untar, remove,* and *copy.* Figures 7 shows user response times or execution times of standard software RAID0, RAID5, and *VC-RAID* including *scd* and *mcd* for both synchronous and asynchronous mode. In the case of synchronous request mode, the performance gains of *mcd VC-RAID* over RAID5 are a factor of 5.0 for untar (Figure 7a), a factor of 16 for remove (Figure 7b), and a factor of 4.2 for copy (Figure 7c) respectively. Even the *scd VC-RAID* presents performance improvements of a factor 2.8, 3.9, and 2.3, respectively for the three programs. It can be seen from these figures that the *VC-RAID* even performs better than standard RAID0. There are three reasons for such a great performance gain. First of all, in this experiment we measured performance of real applications. As we indicated earlier, real applications have strong data locality properties. Cache works only when locality exists. This is true for any cache. A CPU cache would not work if application programs do not have locality properties and evenly access data across entire main memory. This is also true for disk caches that would work and show performance advantages if disk accesses have locality properties. Secondly, there are many overwrites for these metadata intensive workloads such as modifying super block, group descriptors, directory entries. The *VC-RAID* can capture these overwrite operations in the write cache before they go to disks. As a result, frequency of destage operations is reduced significantly in real applications. The final reason is that even with RAID0, small data are written to data disk under this synchronous request mode whereas burst simultaneous small requests are combined into large requests in *VC-RAID*.

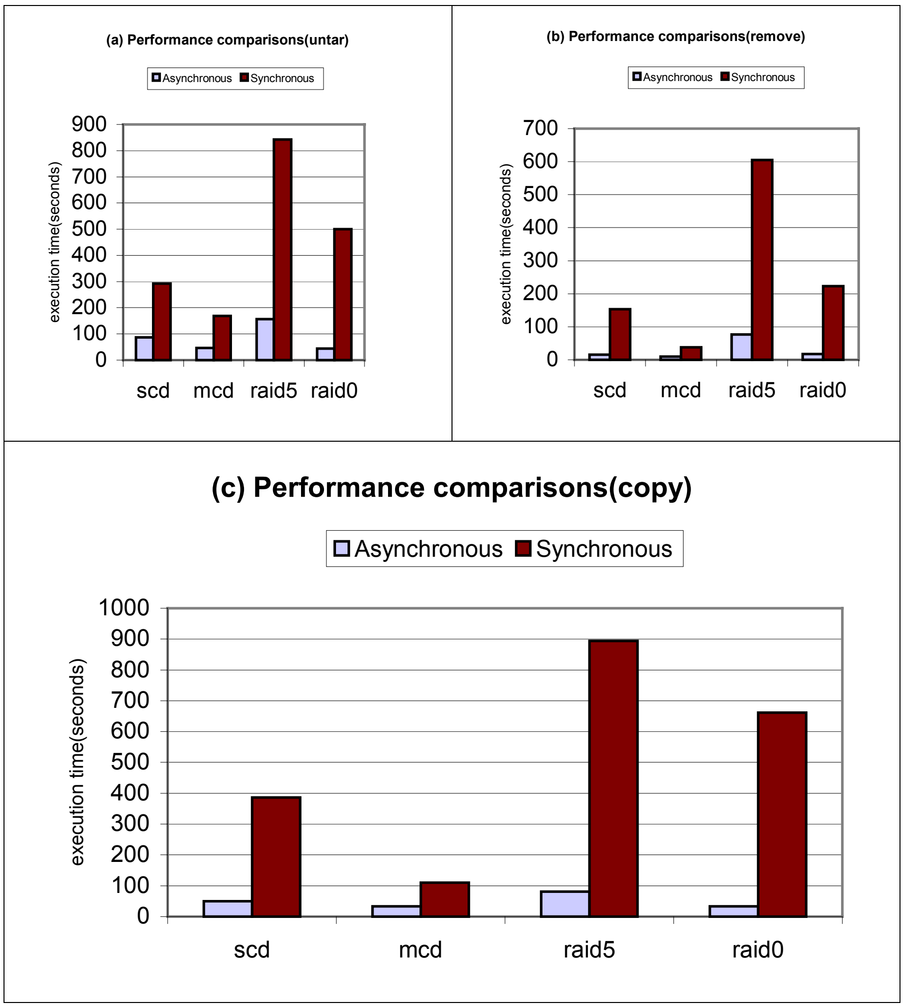

Figure 7. *Results of untar/copy/remove.*

In the case of asynchronous requests, we also observed significant performance gains compared to standard software RAID. The performance improvements range from a factor of 2.5 (Figure 7c) to 7.9 (Figure 7b). This great performance gain can be mainly attributed to the effective and large cache size of the *VC-RAID*. In our test, the total system RAM is 64MB. Taking into account the space used by OS, the available RAM for caching and buffering is about 50MB, 16MB of which is used for system buffering, so the RAM available for file cache is only about 34MB. Even though user process does not have to wait until data are written into disk, it is often blocked because the file cache size is much smaller than data set of the experiment, which is 122MB. However, the cache size of *VC-RAID* is 200MB including RAM and cache disk. This large hierarchical disk cache appears to the host as a large RAM that can absorb quickly a large amount of write requests resulting in great performance gain.

We also noticed that the performance of *VC-RAID* is even much better than that of RAID0 for *remove* program for both synchronous and asynchronous mode (Figure 7b). The reason for this is as follows. When a file or a directory is removed, the super block, group descriptors, block and inode bitmap and corresponding inodes are updated while the “real data” are not removed. In our test, 952 subdirectories and 12322 files are removed, many blocks are overwritten over and over again. Most of these operations are small writes. *VC-RAID* can cache these small writes effectively giving rise to great performance gain.

4.3.3. Bonnie Throughput

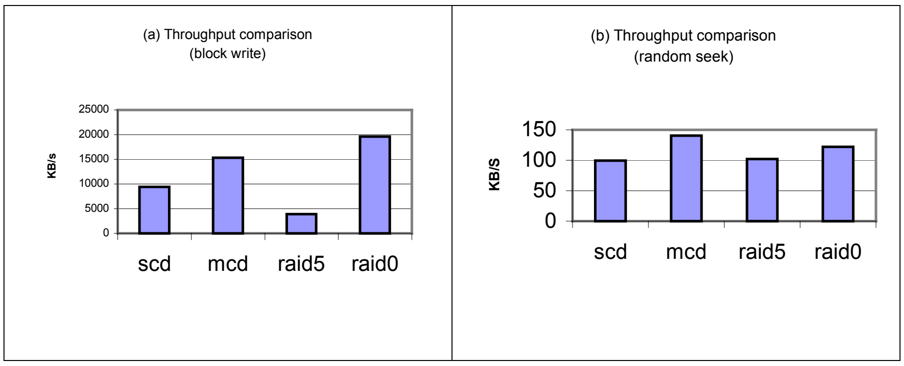

Figure 8. *Results of Bonnie.*

Our next experiment is to compare the throughput of *VC-RAID* to that of RAID0 and RAID5 using the benchmark program Bonnie. The size of the data set is 100M bytes. Figures 8a and 8b show the results for block write and random seek respectively. The performance gains of *scd* and *mcd* over RAID5 are 2.4 and 3.65 for block write as shown in Figure 8a. We also notice that the performance of *mcd* are comparable to that of RAID0, because all requests are write operations under the block write test, where small writes are absorbed by Virtual NVRAM cache. For the random seek operations, we observed very small performance gain of *VC-RAID* over RAID5 as shown in Figure 8b. The reason is that 90% of requests are read requests during the random seek test, which are bypassed to the disk array by *VC-RAID* directly. Furthermore the 10% follow up write operations are in place write with no seek time involved. Therefore, the performance of RAID5, *VC-RAID* and RAID0 are very close (Figure 8b). From Figure 8b we also noticed that the performance of random seeks are very poor, which confirms the claim of the Bonnie author, “random seeks on Unix file systems are appallingly slow [BON 96]”.

4.3.4. PostMark Throughput

Our final experiment is to use PostMark to measure the I/O throughput in terms of transactions per second. PostMark measures performance in terms of transaction rates in the ephemeral small-file regime by creating a large pool of continually changing files. The file pool is of configurable size. In our tests, PostMark was configured in three different ways as in [KAT 99], i.e: 1) small: 1000 intial files and 50000 transactions; 2) medium: 20000 initial files and 50000 transactions; and 3) large: 20000 initial files and 100000 transactions. The read and block sizes are set to 1KB which is the default block size on Linux. We left all other PostMark at their default settings. Our *VC-RAID* was configured by using single cache disk (*scd*). The results of testing are shown in Table 2, where larger numbers indicate better performance.

From these results, we see that *VC-RAID* exceeds the throughput of the built-in software RAID 5 and is comparable to that of RAID 0. The performance improvement of *VC-RAID* over RAID 5 ranges from 67.7% to 110%.

Series	*RAID 0*	*SCD*	*RAID 5*
Small	1111	941	561
Medium	68	63	30
Large	31	28	16

Table 2. *PostMark results in terms of Transactions Per Second.*

5. Related Work

A number of approaches for overcoming small write problems in RAID5 exist in the literature.

Nonvolatile RAM (NVRAM). Modern hardware RAID systems make extensive use of NVRAM write caches to allow fast write [CHE 94][HOU 97][MEN 93][TRE 95], or asynchronous write. Write requests are acknowledged before they go to disk.

Such write caches significantly reduce user response times of RAID. Large write caches can also improve system throughput by taking advantages of both temporal locality and spatial locality of general workloads. While most commercial hardware RAID systems have large write caches for better performance, there is no readily cache solutions for software RAID. *VC-RAID* uses a small RAM and a log disk to form a large and non-volatile write cache.

Striping. A scheme called *parity striping* [GRA 90] proposed by Jim Gray et al has similar performance to a RAID with infinitely large striping unit. Chen and Lee [CHE 95] investigate how to stripe data across a RAID Level 5 disk array for various workloads and studies the optimal striping unit size for different numbers of disks in an array. Jin et al [JIN 98] divided the stripe write operation in RAID 5 into two categories, *full stripe write* and *partial stripe write*. They proposed an adaptive control algorithm to improve the partial stripe write performance by reducing the stripe read/write operations. Mogi and Kitsuregawa proposed a *dynamic parity stripe reorganization* [MOG 94] that creates a new stripe for a group of small writes and hence eliminating extra disk operations for small writes all together. Hua, Vu and Hu improve the *dynamic parity stripe reorganization* further by adding distributed buffer for each disk of RAID [HUA 99]. *AFRAID* [SAV 96] proposed by Savage and Wilkes eliminates the small write problem by delaying parity updates and allowing some stripes to be non-redundant for a controllable period of time.

Logging. Logging techniques are used to cure the small write problem by many researches [CHE 2000][GAB 98][SEL 93][SHI 95][STO 93]. A very intelligent approach called *Parity Logging* [STO 93] proposed by Stodolsky, Holland and Gibson makes use of high speed of large disk access to log parity updates in a log disk. As a result, many parity changes are collected and are written into disk in large sizes, read from disk in large sizes, and compute new parity in large sizes. Another approach is *Data Logging* proposed by Gabber and Korth [GAB 98]. Instead of logging parity changes, it logs the old data and new data of a small write in a log disk and compute parity at later time. Data logging requires 3 disk operations for each small write: reading old data, writing new data in place, and writing a log of old data and new data in log disk, as opposed to 4 operations in RAID5.

Zebra [HAR 95] and *xFS* [AND 95] are both distributed file systems that use striped logging to store data on a collection of servers. *Zebra* combines LFS with RAID to allow for faster disk operation, and a single meta-data server is used. *xFS* eliminates the centralized elements of Zebra that could cause a bottleneck. Both *Zebra* and *xFS* are complete file systems, while our *VC-RAID* is a device level driver.

HP *AutoRAID* [WIL 95] uses a two-level storage hierarchy and combines the performance advantages of mirroring with the cost-capacity benefits of RAID5 by mirroring active data and storing inactive data in RAID5. If the active subset of data changes relatively slow over time, this method has great performance/cost benefits. Similarly, another scheme called *dynamic parity grouping* (DPG) proposed by Yu et al [YU 2000] partitions the parity into special parity group and default parity group. Only the special parity blocks are buffered in the disk controller cache, while the default parity blocks remain on disks.

6. Concluding Remarks

We have presented the design and implementation of a virtual NVRAM cache (*VC-RAID*) for software RAID. We called it virtual NVRAM because it does not have real or physical NVRAM, nor does it need any additional hardware. The virtual NVRAM cache is completely implemented using software while keeping properties of NVRAM. It is nonvolatile because the cache is part of a disk and it is fast due to the use of log structure in cache disks.

A prototype do-it-yourself RAID with a large virtual NVRAM cache (*VC-RAID*) has been implemented under the Linux kernel. Through different benchmark tests, *VC-RAID* demonstrates superb performance advantages over the standard built-in software RAID5 shipped with the Linux package. Depending on the workload characteristics, performance gains due to *VC-RAID* range from 67.7% to an order of magnitude. For applications that have data locality, we observed up to a factor of 16 performance improvements in terms of user response time. In many situations, *VC-RAID* achieves similar performance as RAID0 and some time better than RAID0 indicating that *VC-RAID* realizes close to maximum potential to hide small write problems in RAID5.

We are currently working on building a *VC-RAID* with mirrored cache disks to further improve the reliability. We will also be working on porting *VC-RAID* on hardware RAID controllers, where *VC-RAID* will make use of controllers' RAM other than the host system RAM.

Acknowledgements

This research is supported in part by National Science Foundation under grants MIP-9714370 and CCR-0073377. The authors would like to thank the anonymous reviewers for their many helpful comments and suggestions.

References

[AND 95] Anderson T., Dahlin M., Neefe J., Patterson D., Roselli D., and Wang R., "Serverless Network File Systems", In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Dec. 3-6, 1995, p.109-126.

[ASA 95] Asami S., Talagala N., Anderson T., Lutz K., and Patterson D., “The Design of Large-Scale, Do-It-Yourself RAIDs”, URL: *Http://www.cs.berkeley.edu/~pattrsn/papers.html,* 1995.

[BON 96] Bonnie benchmark, “Bonnie benchmarks v2.0.6”, URL: *http://www.textuality.com/bonnie/,* 1996.

[CHE 94] Chen P., Lee E., Gibson G., Katz R., and Patterson D., “RAID: High-Performance, Reliable Secondary Storage”, *ACM Computing Surveys 26(2)*, 1994, p.145-185.

[CHE 95] Chen P., and Lee E., “Striping in a RAID Level 5 Disk Array”, *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modelling of computer systems*, May 15-19,1995, p.136-145.

[CHE 2000] Chen Y., Hsu W., and Young H., “Logging RAID - An Approach to Fast, Reliable, and Low-Cost Disk Arrays”, *Euro-Par*’2000, p.1302-1312.

[CHU 92] Chutani S., Anderson O., Kazar M., Leverett B., Mason W., and Sidebotham R., “The Episode File System”, *Winter USENIX Conference*, Jan. 1992, p. 43-60.

[COO 96] Coombs D., “Drawing up a new RAID roadmap”, *Data Storage*, vol.3, Dec. 1996, p.59-61.

[GAB 98] Gabber E., and Korth H., “Data Logging: A Method for Efficient Data Updates in Constantly Active RAIDs”, *Proc. of the 14th Intl. Conference on Data Engineering, 1998*, p.144-153.

[GRA 90] Gray J., Horst B., and Walker M., “Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput”, *In Proc. Of the 16th Very Large Database Conference (VLDB)*, 1990, p.148-160.

[GRA 93] Gray J., and Reuter A., *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993.

[HAR 95] Hartman J., and Ousterhout J., “The Zebra Striped Network File System”, *ACM Transactions on Computer Systems*, vol.13 no.3, 1995, p.274-310.

[HOU 97] Hou R., and Patt Y., “Using Non-Volatile Storage to Improve the Reliability of RAID5 Disk Arrays”, *the 27th Annual Intl. Symposium on Fault-Tolerant Computing*, 1997, p.206-215.

[HU 96] Hu Y., and Yang Q., “DCD-disk caching disk: A New Approach for Boosting I/O Performance”, *23rd Annual Intl. Symposium on Computer Architecture*, Philadelphia PA, May, 1996, p.169-178.

[HU 99] Hu Y., Yang Q., and Nightingale T., “RAPID-Cache: A Reliable and Inexpensive Write Cache for Disk I/O Systems”, *Proc. of the 5th Intl. Symposium on High Performance Computer Architecture*, Jan. 1999, p. 204-213.

[HUA 99] Hua K., Vu K., and Hu T., “Improving RAID Perfromance Using a Multibuffer Technique”, *Proc. of the 15th Intl. Conference on Data Engineering*, 1999, p. 79-86.

[JIN 98] Jin H., Zhou X., Feng D., and Zhang J., “Improving Partial Stripe Write Performance in RAID Level 5”, *Proceedings of 2nd IEEE International Caracas Conference on Devices, Circuits and Systems*, March 2-4, 1998, Margarita, Venezuela, p.396-400.

[KAT 99] Katcher J., "PostMark: A New File System Benchmark", Technical Report TR3022, Network Appliance, *http://www.netapp.com/tech_library/3022.html,* 1999.

[MAT 97] Matthews J., Roselli D., Costello A., Wang R., and Anderson T., "Improving the Performance of Log-Structured File Systems with Adaptive Methods", *Proc. Sixteenth ACM Symposium on Operating System Principles*, Saint Malo, France, October 5-8 1997.

[MCK 99] McKusick M., and Ganger G., "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem", *Proc. Of the FREENIX Track: 1999 USENIX Annual Technical Conference*, June 6-11, 1999.

[MEN 93] Menon J., and Cortney J., "The Architecture of a Fault-Tolerant Cached RAID Controller", *Proc. of the 20th Annual Intl. Symposium on Computer Architecture*, May 1993, p. 76-86.

[MOG 94] Mogi K., and Kitsuregawa M., "Dynamic parity stripe reorganization for RAID5 disk arrays", *Proc. Of the 3rd Intl. Conference on Parallel and Distributed Information Systems*, Sept. 1994, p. 17-26.

[MOH 95] Mohan C., "Disk Read-Write Optimizations and Data Integrity in Transaction Systems Using Write-Ahead Logging", *Proc. 11th International Conference on Data Engineering*, March 1995.

[RUE 93] Ruemmler C., and Wilkes J., "UNIX Disk Access Patterns", *Proc. of the USENIX'93 Winter Conference,* San Diego, CA, Jan.1993, p. 405-420.

[SAV 96] Savage S., and Wilkes J., "AFRAID – A Frequently Redundant Array of Independent Disks", *Proc. of 1996 USENIX Annual Technical Conference,* Jan. 22-26,1996, p. 27-39.

[SEL 93] Seltzer M., Bostic K., McKusick M., and Staelin C., "An Implementation of a Log-Structured File System for UNIX", *Winter USENIX Proceedings*, Jan. 1993, p. 201-220.

[SER 2000] Server bench 4.1, URL: *http://www.zdnet.co.uk/pcmag/labs/2000/07/os/22.html*, July 2000.

[SHI 95] Shirriff K., "Sawmill: A Logging File System for a High-Performance RAID Disk Array", *Technical Report, CSD-95-862*, Computer science department, University of California at Berkeley, 1995.

[STO 93] Stodolsky D., Holland M., and Gibson G., "Parity Logging: Overcoming the small write Problem in Redundant Disk Arrays", *Proc. of the 21th Annual Intl. Symposium on Computer Architecture*, 1993, p. 64-75.

[TRE 95] Treiber K., and Menon J., "Simulation study of cached RAID5 designs", *Proc. Of Intl. Symposium on High Performance Computer Architectures*, Jan. 1995, p. 186-197.

[WIL 95] Wilkes J., Golding R., Staelin C., and Sullivan T., "The HP AutoRAID Hierarchical Storage System", *Proc. Of the Fifteenth ACM Symposium on Operating System Principles*, Dec. 3-6, 1995, p. 96-108.

[YU 2000] Yu P., Wu K., and Dan A., "Dynamic Parity Grouping for Efficient Parity Buffering for RAID-5 Disk Arrays", *Computer Systems Science and Engineering*, vol. 15 No. 3, May 2000, p. 155-163.