# *Achieving Predictable Response Time with an Intelligent File System Updater**

Li-Chi Feng and Ruei-Chuan Chang
National Chiao Tung University

---

ABSTRACT: Delayed write is a very popular technique to improve the file system performance of UNIX operating systems. When write operations are delayed, an update policy decides how and when to write these modified blocks to their assigned disk locations. Recent research results show that conventional update mechanisms do not perform very well, because they neglect the fact that different I/O requests naturally have different characteristics. It makes time critical interactive jobs endure large variations in response time. Update policy controls background write activities that are less time critical and should be performed under constraints that do not violate the urgency of other time critical jobs.

In this paper we propose an effective update scheme. We suggest new techniques: *burst declustering* and *opportunistic asynchronous write*. Besides, we develop a system activity sensor called *SAPRO* (system activity probe) that monitors the system activities and disk queuing status to adjust the behavior of our algorithm. Performance evaluation shows that our algorithm can alleviate the lengthy queuing delay, reduce the variance and worst case read response time significantly (30% and 51% respectively). The mean read response time and total system performance are also improved.

---

*This research was sponsored by the National Science Council of R.O.C. under grant NSC83-0408-E009-055

## 1. Introduction

Typical file systems use disk cache to reduce average access time for data storage and retrieval [Karedla et al. 1994, Smith 1985]. Delayed write used in UNIX operating systems is a very popular technique to improve file system performance [Bach 1986]. When a user program makes a write system call to write some data, the data are not written to disk immediately. Instead, the modified data are kept dirty in memory for a while. It hopes that when the write operations are delayed, subsequent requests will rewrite these cached locations frequently. Thus a large number of disk I/Os are eliminated.

When the write operations are delayed, we must decide how and when to write these modified blocks to their disk locations. If there is no bound on the delay period, a system crash will cause serious data loss. The algorithm used to decide how and when to write delayed data to disk is referred as an update policy. Conventional UNIX systems use a periodic update (PU) policy, writing all delayed-write data once every 30 seconds [Bach 1986].

Recent analytical and simulation results, presented by Carson and Setia [Carson & Setia 1992], showed that the PU policy actually performs less well in many cases than the write-through (WT, making all writes immediately) policy. The bulk arrivals generated by the periodic update policy cause a traffic-jam effect which results in severely degraded service. Ruemmler and Wilkes also noted that bursts of delayed writes caused by periodic update policy can seriously degrade performance [Ruemmler & Wilkes 1993].

Carson and Setia proposed several alternative write policies. The first, use the WT (write through) policy with all writes performing asynchronously. A more attractive alternative is the *periodic update with read priority* (PURP) policy which gives higher priority to read operations. A third alternative is the *individual periodic update* (IPU) policy, in which write operations are delayed for a fixed time interval on an individual basis, rather than being presented to the disk system in bulk [Carson & Setia 1992].

Fixed-priority schemes such as PURP introduce the potential for infinite delays of delayed writes, if read load is heavy enough to saturate the disk [Carson & Setia 1992]. Peacock also reported that adding PURP to System V Release 4,

in which the buffer cache can be quite large, actually reduces benchmark performance by preventing asynchronous requests from getting a sufficient share of the disk [Peacock 1992].

Mogul [Mogul 1994] used implementation to validate Carson and Setia's results on an actual system. He found that IPU adds little code to the kernel and performs much better than periodic update, but IPU depends on a relatively uniform distribution of file writes over time to achieve its higher performance. According to user level file access patterns, file system activity is bursty [Baker et al. 1991, Ousterhout et al. 1985]. So, IPU policy cannot avoid long disk queue in general.

As more and more write operations are delayed by various delayed write mechanisms [Bach 1986, Ganger & Patt 1994, Feng & Chang 1994], read operations become more and more time critical than before. Due to the popularity of interactive computing, the read response time and especially the variance in read response time are becoming the most important performance metrics.

Carson and Setia showed that how write requests are presented to the disk system is more important than the write request rate if read response time is the most important performance metric [Carson & Setia 1992]. As the size of delayed write buffer rapidly increases, a better update policy is critical to system performance [Mogul 1994].

In this paper we propose an intelligent file system updater that uses a system activity sensor called *SAPRO* (system activity probe) and the following two techniques to achieve an effective update policy: *burst declustering* and *opportunistic asynchronous write*. Burst declustering is a technique for detecting and breaking up potential write bursts. Opportunistic asynchronous write is a mechanism for using I/O idle periods to accomplish less critical write operations. *SAPRO* can observe the current system status to adjust the behavior of our update policy. Performance evaluation shows that our algorithm significantly reduces the variance and worst case read response time.

The rest of this paper is organized as follows. In Section 2, we introduce the new scheme. The design and implementation of our facility are described in Section 3. Performance evaluation is given in Section 4. Several related works are reviewed in Section 5. Concluding remarks are given in Section 6.

## 2. *Proposed Scheme*

The main reason for the inadequate performance of PU is its lengthy queuing delays. By dumping all the dirty blocks onto the disk at once, PU generates a long queue. In some UNIX file systems, this long queue causes the latency-sensitive synchronous operations, such as file reads or synchronous writes, to be forced to

wait behind latency-insensitive operations in the queue [Mogul 1994]. Effective use of disk caching and disk scheduling can alleviate this problem, but only under a narrow range of operating conditions [Carson & Setia 1992].

Although IPU is better in most cases, it cannot avoid the long disk queue in general. If the file writes themselves arrive in bursts, the benefits of IPU are eliminated. In the worst case, IPU and PU have the same performance [Mogul 1994].

Traditional I/O system studies treat all I/O requests as equally important. This is not true; different I/O requests naturally have different characteristics. Synchronous requests have the side effect of blocking the calling process, delayed writes can wait longer and allow more flexibility in choosing the proper time for updating. Ganger and Patt proposed a process flow model and claimed that I/O research should take a global view that takes overall system performance into consideration [Ganger & Patt 1993]. They also emphasized the importance of I/O requests classification. In order to achieve higher system performance and smaller variations in response time for interactive jobs, non-critical requests should not interfere with the completion of time critical requests if possible.

Update policy controls the background activities that flush out the delayed-write cache (buffer). Basically, these background write activities are less time critical and should be performed under constraints that do not violate the urgency of other time critical jobs. This goal can be achieved if we can maintain a short disk queue. Thus, subsequent synchronous (time critical) operations can be served as soon as possible. Because disk queue is the only path to accomplish the disk I/O requests, subsequent synchronous requests will endure long delays or large variations in response time if the queue length is very long or the requests are bursty. This will degrade the system performance.

In order to ensure a reasonable disk queue length, we propose two techniques: *burst declustering* and *opportunistic asynchronous write*. Burst declustering is a technique that detects and breaks up potential write bursts. If many clustered updated blocks are found, the algorithm will decluster these incoming write operations over time. Opportunistic asynchronous write is a mechanism that senses the disk queuing status and utilizes I/O idle periods to accomplish less time critical nonsynchronous write operations.

### *2.1. Burst Declustering*

In order to explain this algorithm, we assume that our system has a linear buffer pool that contains all system buffers. Each buffer in the pool has the following possible states: clean or dirty, free or in-use. The declustering algorithm linearly scans the entire buffer pool at some specified frequency. The array `age` is used

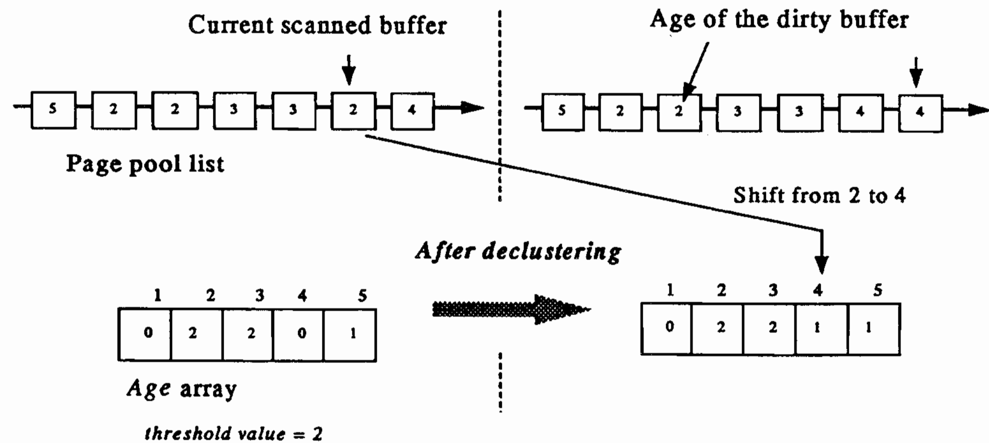



Figure 1. Declustering operation.

to record the aged situation of the dirty buffers in the pool. For example, `age[i]` represents the number of dirty buffers that have been found that are $i$ seconds old since last reset of the `age` array. The reset time of `age` array is decided by the implement. The age of a dirty buffer is the elapsed time in seconds since the buffer is first modified.

During the buffer pool scanning, we calculate the age of each dirty buffer and update the statistics kept in the `age` array. If too many (no less than a threshold value) dirty buffers are clustered in the same age (e.g. are $i$ seconds old) as the buffer currently being scanned, the declustering operation is executed, otherwise, they would cause large write burst in the near future. The algorithm begins a linear scan from `age[i]` to `age[i+declustering_range]`, and shifts the age of the dirty buffer currently being scanned forward to the nearest age with `age` array entry value smaller than the threshold. If all entries in the declustering range have values greater than or equal to the threshold, the entry with the lowest value is chosen. This technique can prevent too many dirty buffers from being flushed out at the same time. `declustering_range` is a control parameter that prevents shifting dirty buffers too far away. It will be explained later.

Figure 1 is a small example used to demonstrate the declustering operation. On the left side of the graph, one cell marked with a vertical arrow is the currently scanned buffer. The original age of this buffer is 2. When we look at the `age` array we find that `age[2]` and `age[3]` are full (no less than the threshold value), so we shift its age to 4 as illustrated in the right side of the figure.

### 2.2. *Opportunistic Asynchronous Write*

In modern file systems, many noncritical write operations are delayed or carried out asynchronously. From the point of view of overall system performance, these write operations are less time critical. Priority should be given to synchronous requests under constraints that do not violate the lifetime limits of dirty buffers belonging to noncritical writes. The flush rate of delayed operations can be controlled by the file system flush daemon, but no mechanism can be used to control those asynchronous write operations in conventional UNIX systems.

Process execution normally contains a mixture of CPU processing and I/O operations. The bursty access patterns reported in research literature suggest that, a system should have some periods of low I/O (disk) utilization. If we can use these I/O idle (low utilization) periods to accomplish some of the background write activities, the overlap between CPU processing and I/O operation should be increased. This will smooth down the write load of I/O subsystem and improve the system performance.

*Opportunistic asynchronous write* is the technique for adjusting the asynchronous write flush rate. We use disk queue status reported by *SAPRO* as the threshold for achieving opportunistic asynchronous write. When the I/O system load is too heavy, asynchronous write operations issued by the operating system are queued to OAW (Opportunistic Asynchronous Write) queue to keep the disk queue short. These queued operations will be flushed to disk queue later when an I/O idle period is detected. This can help to alleviate the problem due to lengthy queuing delay.

## 3. *Design and Implementation*

In this section we describe the design and implementation of our update policy. UNIX SVR4/MP Version 2 operating system is used as the basis for development of our algorithm. We first introduce the original scheme used by SVR4/MP, and then show how we improve the original scheme to accomplish our goal.

### 3.1. *Original Scheme of SVR4/MP Version 2*

There are two tunable parameters in SVR4/MP that the system administrator can use to control the operation of the update mechanism [UNIX 1990]:

`FSFLUSHR`: This is the file system flush rate. It specifies the rate in seconds for checking (by a file system checker called `fsflush` daemon) the need to

write the file system buffers, modified inodes, and mapped file pages to disk. Mapped file page means those memory pages used as I/O cache, it will be explained later. The default value of FSFLUSHR is one second, it means that `fsflush` will be awaked every second. If the `fsflush` finds some aged dirty buffers, it will update them to disk. A dirty buffer is aged if it is dirty and has been memory resident beyond the limit specified by the NAUTOUP parameter.

NAUTOUP: This parameter specifies the buffer age in seconds for automatic file system updates. System buffers and other cached file attributes (metadata, such as inodes) are written to the hard disk when they have been memory-resident for the interval specified by this parameter. Specifying a small NAUTOUP value increases system reliability by writing the buffers to disk more frequently and decreases system performance. Specifying a large value increases system performance at the expense of reliability. The default value is 60 seconds.

In order to reduce the overhead caused by the file system checker, it is not required to check the entire page pool list for each invocation of `fsflush` if FSFLUSHR is high enough. In the implementation of SVR4/MP, only

*(Total number of pages in the page pool list)*∗FSFLUSHR/NAUTOUP

pages need to be checked for each invocation of `fsflush`.

Buffer cache is a pool of internal data buffers that contain data from recently used disk blocks. In conventional UNIX systems, buffer cache is the only path for block I/O [Bach 1986]. Due to the increasing gap between disk and CPU speed, it is a critical component in determining file system performance. Recently, many modern operating systems use mapped file pages for I/O cache. This has the advantage of using all available memory as I/O cache or memory cache as needed. Thus, the importance of conventional buffer cache has been reduced. In SVR4/MP, the buffer cache is used to access some metadata only. The position of the buffer cache module is illustrated in Figure 2.

Inode cache is also a pool of internal buffers used to store recently accessed inodes. Inode is the data structure in which metadata about a file are stored. Kernel needs the information stored in the inode to access file data. Recent research results show that metadata I/O account for a large number of all disk I/Os [Muller & Pasquale 1991, Ruemmler & Wilkes 1993]. So, an efficient inode cache is also important to system performance.

SVR4/MP Version 2 uses a hybrid update policy. For buffer cache and inode cache, it uses IPU policy. Every time the `fsflush()` is awaked, it scans the entire buffer cache and writes those aged entries (lifetime greater than NAUTOUP) to

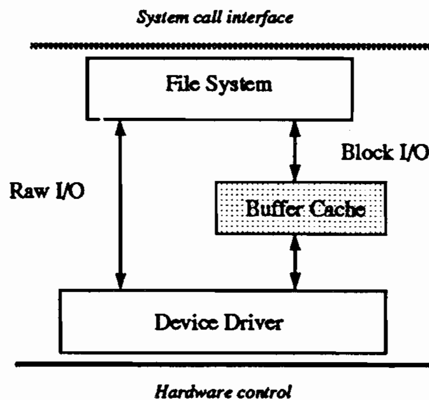



Figure 2. Position of the buffer cache.

disk. It will also `flush` out all dirty inode cache entries to disk every `NAUTOUP` seconds. For mapped file page, PU policy with a one-second period (`FSFLUSHR` is set to 1 second) is used. `fsflush()` checks each page in the page pool once every `NAUTOUP` second, and writes all mapped file pages found dirty out to disk. This means that only one-sixtieth of the page pool is checked each time `fsflush()` is awaked if `NAUTOUP` is set to 60 seconds. This will alleviate the write-burst problem caused by the traditional 30 seconds PU policy [Carson & Setia 1992]. Because PU based policies ignore the age of dirty pages, they may *flush* out many *fresh* dirty pages.

### *3.2. The Approach*

In light of recent research results, and file system consistency requirements, we use IPU policy as the basis of the new system. Our first step is to change SVR4/MP to a pure IPU policy. IPU is a time-based update policy where each dirty block lives in memory for a prespecified time limit. We add one timestamp field to the software page structure to record the time-of-first-modification of every dirty page. The software page structure is used to maintain the identity and status of each physical page. Then, we change the code accomplishing the page pool scan to write out only those mapped file pages that are dirty and aged (have been memory-resident for `NAUTOUP` seconds). After that we implement our declustering algorithm and the mechanism that performs opportunistic asynchronous write. The value of the global variable `lbolt` (which keeps track of the number of ticks since last boot) is used as the timestamp. Thus the age of a dirty page can be calculated as follows:

*age in ticks* = `lbolt` - *timestamp in the page structure*

### 3.3. *Declustering Mechanism*

As described in previous paragraphs, IPU depends on a relatively uniform distribution of file writes to achieve higher performance. But if an application's access behavior is bursty, IPU's advantage is lost. In order to fix this problem, we add a burst declustering algorithm to the mapped file page scan in the `fsflush` daemon. Because the size of the mapped file pages can be extended to occupy large portions of available memory, they will be the source of most background write activities.

#### 3.3.1. *Local Declustering*

In order to alleviate the write-burst problem caused by background write activities, modern operating system update policies adopt an approach that increases scanning frequency and reduces the range of each scan. For example, in SVR4/MP's original scheme, `fsflush` daemon is awaked once per second but scans only one-sixtieth of the page pool during each run. If there are many dirty pages clustered around some age but evenly distributed across different scan periods, it is not a real potential burst and does not hurt system performance. This kind of clustering is called *false-clustering* and is caused by the implementation of the update mechanism.

Our declustering mechanism is a direct implementation of the algorithm described in Section 2. Besides, the point of view is changed from global to local to avoid declustering those *false-clustered* dirty pages. This means that only dirty pages clustered in the same scan period are declustered. The `age` array is reset (set to 0) every time the `fsflush` is awaked.

The construction of the `age[]` array is embedded in the page pool scan in the `fsflush` daemon. When the age of a dirty page is checked by `fsflush`, its corresponding entry in the `age` array is updated (increased). The declustering operation is executed if a great many clustered dirty pages are found.

#### 3.3.2. *Declustering Range and Threshold*

One potential problem is some artificial burstiness caused by the declustering algorithm. In order to ensure the file system hardening requirement, we can only increase the age of these clustered pages (one way declustering). Without some bound on declustering, too many 60-second old pages will be generated. Thus, a large burst will be inevitable. So, a localization technique is used. Now we use a fixed `declustering_range` (e.g. 10 seconds). If a potential burst (too many clustered pages) is found at age $i$, we try to decluster within the range from age $i$ to age $i +$ `declustering_range` ($i + 10$).

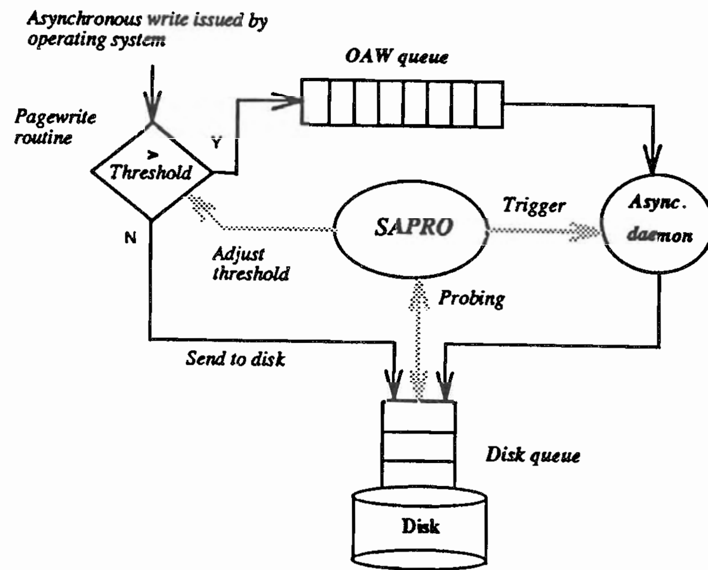



Figure 3. Opportunistic asynchronous write mechanism.

A threshold is used to control the declustering operation. Smaller values can decluster the `age[]` array more finely, but also require more computation time. A dynamically adaptable threshold sounds attractive but may entail larger overhead. Because our declustering algorithm is linearly bounded by the `declustering_range` parameter of the algorithm, a fixed declustering threshold is used in the current implementation.

### *3.4. Accomplishing Asynchronous Write Opportunistically*

The opportunistic asynchronous write mechanism is illustrated in Figure 3. When an asynchronous write operation is issued by the operating system, the control is passed to the page write routine (`vop_putpage`). The routine has been modified to check the disk load before sending the I/O request to the disk queue. If the current disk load is beyond the control threshold reported by *SAPRO*, the asynchronous write is delayed and inserted into the OAW (opportunistic asynchronous write) FIFO queue. *SAPRO* monitors the queuing status of the disk constantly and adjusts the control threshold accordingly. When a low I/O-load period is found, it triggers the *asynchronous write daemon* to dequeue asynchronous write requests from the *OAW* queue and send them to the disk queue. *Asynchronous write daemon* is a system daemon process designed by us and is responsible for flushing delayed asynchronous writes.

The most important element of the opportunistic asynchronous writing is detecting low I/O utilization periods. Although an absolute control threshold is easy to implement, a continuous stream of large I/O operations may keep the system busy for long periods of time. Thus, delayed asynchronous writes must endure long delays. A large number of system resources are also required to describe these delayed operations.

Because of file system consistency requirements, we cannot delay asynchronous writes too long, so, a relative control threshold is used. In the current implementation, *SAPRO* monitors the disk queue and calculates the average disk queue length for the previous 10 seconds (`avq10s`) as the control threshold. When an asynchronous write is coming, the page write routine reads the current disk queue length from the system *disk* structure. If current disk queue is shorter than or equal to `avq10s`, we consider the I/O system to be in a relatively idle (low utilization) state, and the I/O request is issued as usual. Otherwise the I/O request is put into the *OAW* queue.

### *3.5. System Activity Probe (SAPRO)*

We design and implement a facility called *SAPRO* (*system activity probe*) to monitor the system status: CPU utilization, disk queue average waiting time, and current disk queue length. But only the disk queuing status is currently being used. *SAPRO* calculates and reports the last 10 second average disk queue length (`avq10s`) every ten seconds. `Avq10s` is used as the control threshold in accomplishing opportunistic asynchronous write. If the computed value is too small, a pre-specified minimum value is used. *SAPRO* also compares the current disk queue length with `avq10s` every second and triggers the *asynchronous write daemon* to write queued operations when a relatively idle period is found. Thus the flushing behavior of our update algorithm can be adjusted and can adapt to the system status change.

SVR4/MP has many kernel data structures that contain various cumulative data about system activities since last boot. These data are collected during system execution. For example, in the SCSI disk driver send routine, the system records the total number of read/write requests in a structure called `scsi_iotime`. *SAPRO* uses these raw data to calculate the required statistics to control the update algorithm. In current implementation, *SAPRO* adheres to the `fsflush` daemon as it will be awaked once per second.

# 4. *Performance Evaluation*

In order to assess the performance behavior of various update schemes, we have done extensive performance evaluation. In the following paragraph, we shall introduce our experimental apparatus, the benchmarks used, and the performance results obtained from various update policies.

## 4.1. *The Evaluation Environment*

The testbed consisted of a personal computer with a 120 Mhz AMD 486DX4 processor. It contained 16 megabytes of main memory, and two SCSI disks. The first disk was a Quantum LPS540S that held the root file system and the swap partition. The second disk, a Seagate ST31230N, was the test (data) disk. It had a 300 MB test partition and 182.3MB of them are free. The SCSI host adaptor was an Adaptec 1542B. Our update policy was installed through kernel modification. The system was running the UNIX SVR4/MP Version 2 operating system and all measurements were taken without network attachments. Before each test, we *unmount* and *remount* the test partition to eliminate the effects of memory caching. All file systems ran UFS, the block size was 4 kilobytes. The geometry and performance characteristics of the tested disks are illustrated in Table 1.

## 4.2. *Random Read/Write Test*

We perform the random read/write tests similar to the method described in [Mogul 1994]. The read/write load generators proceed in a controlled fashion such that data are read from the disk and written to the in-memory cache. So the reader process is more time critical than the writer process. The purpose of these tests is to show how the new scheme affects the mean, worst case and especially the variance in read response time.

The test programs consist of one reader process and several writer processes. The reader process issues 10000 random read requests (each 1024 bytes long) from a large (32 MB) file. Because the file size is much larger than the main memory size, most random read requests will generate synchronous disk reads to get the required data. If the disk queue length is large or changeable, the read response time will also endure a long delay or a large variance. The purpose of the writer processes is to pollute the in-memory cache and keep the `fsflush` daemon busy to generate enough background write activities to interfere with the read requests.

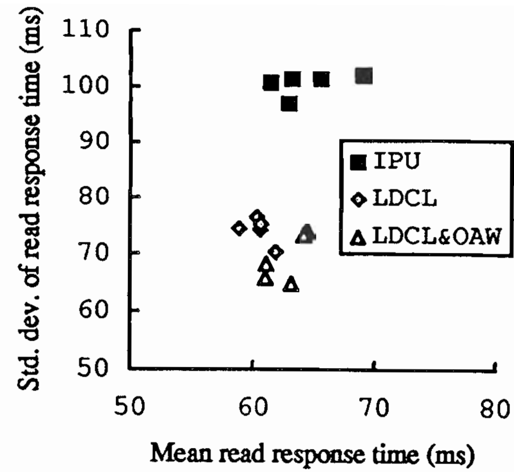
| | Disk1 | Disk2 |
|---|---|---|
| Disk Model | Quantum LPS540S | Seagate ST31230N |
| Interface | SCSI-2 | Fast SCSI-2 |
| Capacity(MB) | 541.4 | 1050 |
| Cylinders | 2740 | 3892 |
| Heads | 4 | 5 |
| Sectors/Track | 125 | 104 |
| Bytes/Sector | 512 | 512 |
| Spindle RPM | 4500 | 5411 |
| Tracks/Zone | 4 | 5 |
| Data Access Time (ms) | 10.7 | 9.4 |
| Transfer Rate (KB/sec) | 2110 | 2520 |

Table 1. The geometry and performance characteristics of the test disks on the test machine (486 processor, ISA bus). The geometry data are reported by the `scsicntl` program from Adaptec Corporation. The performance data are reported by the QBench benchmark from Quantum Corporation. The workload used during the transfer rate test are 65% sequential, 35% random, 60% read, and 40% write. Block size ranges from 1 to 128 sectors.

The function of the writer process is to create a small file, then randomly write 1 KB blocks to this new file. Because the file is new, write to a new block does not need an additional read. Because the file is small, subsequent writes will hit in the memory. So, most disk operations caused by the writer process is coming from the `fsflush` daemon.

Figure 4 shows the results of the random read/write tests. LDCL represents the policy with local declustering. LDCL&OAW represents the LDCL extension with opportunistic asynchronous write. The figure shows a point for each trial (10000 random reads), plotted with mean read response time on the horizontal axis and the standard deviation of read response time on the vertical axis. In these tests the declustering threshold is 6 and declustering range is 10 seconds.

In Figure 4a, the write load is smaller and most background write activities are issued by the `fsflush` daemon. So the LDCL scheme performed well. Compared with IPU policy, LDCL shows large improvements (35.6% in variance, 67% in worst case, and 6.6% in mean read response time). Although LDCL&OAW shows only a 2.6% improvement in mean read response time due to the overhead



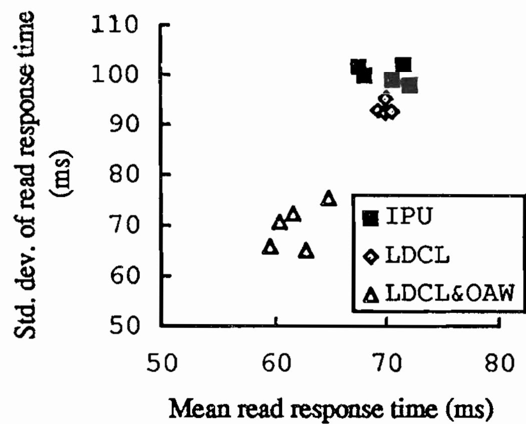




Figure 4. Results of the random read/write test. We perform five trials for each policy. LDCL represents the local declustering policy. LDCL&OAW represents the LDCL extension with opportunistic asynchronous write. The figure shows a point for each trial (10000 random reads), plotted with mean read response time on the horizontal axis and the standard deviation of read response time on the vertical axis.

| | Workload | 1 | | | 2 | | |
|---|---|---|---|---|---|---|---|
| | | Mean | Stdev | Worst | Mean | Stdev | Worst |
| IPU | value | 64.34 | 100.45 | 2890 | 69.94 | 99.88 | 2800 |
| LDCL | value | 60.38 | 74.08 | 1730 | 70 | 93.01 | 2050 |
| | % | 6.56 | 35.6 | 67 | 0 | 7.39 | 36.59 |
| LDCL& | value | 62.7 | 69.15 | 1320 | 61.81 | 69.8 | 1460 |
| OAW | % | 2.62 | 45.26 | 118.94 | 13.15 | 43 | 91.78 |

Table 2. Results of the random read/write test. The table shows the mean, standard deviation, and the worst case read response time (in millisecond) of the various update policies under two different workloads as in Figure 4. The percentages of improvement compared with IPU policy are also shown.

of OAW, it also has 45% and 119% improvements in, respectively, variance and worst case read response time.

When the write load is increased, various asynchronous writes are issued by the memory manager to reclaim memory resource quickly. These write operations are beyond the control of `fsflush` daemon but can be managed by the opportunistic asynchronous write mechanism. Although LDCL does not show good performance in Figure 4b, LDCL plus OAW achieve 11.6%, 30%, and 47.8% improvements in mean, variance, and worst case read response time. The results are summarized in Table 2.

In order to demonstrate the effects of our burst declustering algorithm, we measured the write burst distribution in executing the random read/write test. The results are shown in Figure 5. A 5k bytes kernel trace buffer is used to keep the histogram of the number of blocks queued for disk on each invocation of `fsflush()` daemon. After each run, the traced data are read out through the `/dev/kmem` interface. As shown in Figure 5, most write burst are smaller than 100 blocks under LDCL scheme. But there are a large number of writes with burst size larger than 100 under IPU policy. The local declustering policy (LDCL) indeed smooths down burstiness as compared to the IPU policy.

### *4.3. AIM Benchmarks*

To assess the effectiveness of our scheme on system performance, AIM benchmark is used. AIM Suite III is a synthesis benchmark developed by AIM Tech-

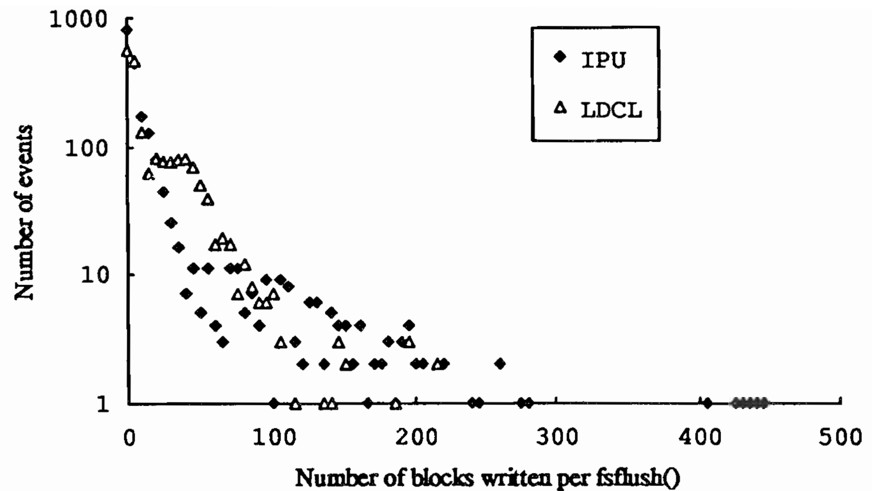



Figure 5. Burst-size distributions for writes during random read/write test. The results are extracted during the execution of workload 1 in Figure 4. Most write bursts are smaller than 100 blocks under LDCL scheme. A large number of writes with burst size larger than 100 under IPU policy.

nology to test the total system performance of all major system components in a multitasking environment. It attempts to simulate the load that a specified number of users would exert on a computer system by running a set of functional benchmarks intended to model a particular application. It contains many primitive functions, and users can adjust the job mix and weight of them. AIM III can be configured to represent various workloads.

We compare the performance of our scheme and IPU update policy using the AIM III benchmark. The test results reported are based on the *standard job mix*, which is designed to describe a typical UNIX environment. The results are illustrated in Figure 6.

In Figure 6, IPU means the IPU policy. LDCL&OAW is the policy proposed by us. As illustrated in Figure 6 and Figure 7, as much as a 4.6% performance improvement is obtained as compared to the IPU policy.

We also measured the write burst distribution while executing the AIM III benchmark. The results are shown in Figure 8. All write bursts are smaller than 108 blocks under our scheme. But there are a large number of writes with burst size larger than 250 under IPU policy. Our update policy obviously alleviates the write burst.

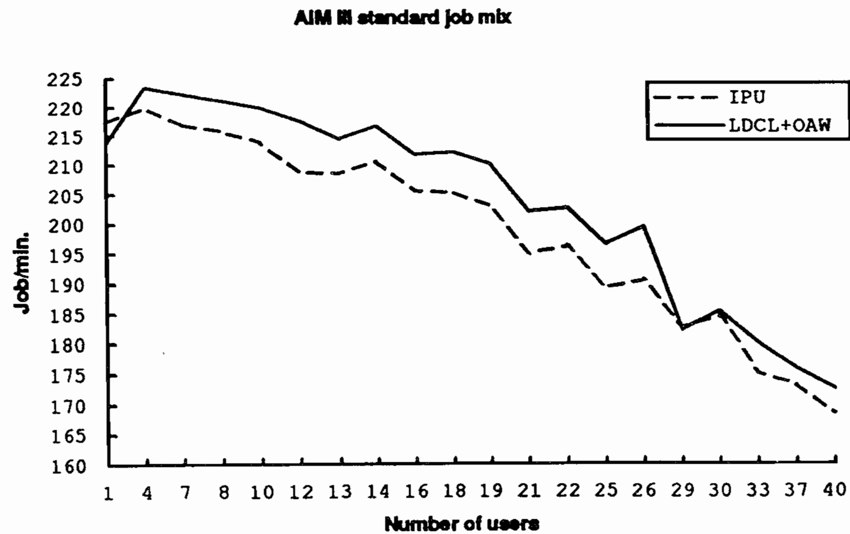



Figure 6. Result of AIM III benchmark. The test results reported are based on the *standard model*, which purports to describe a typical UNIX environment.

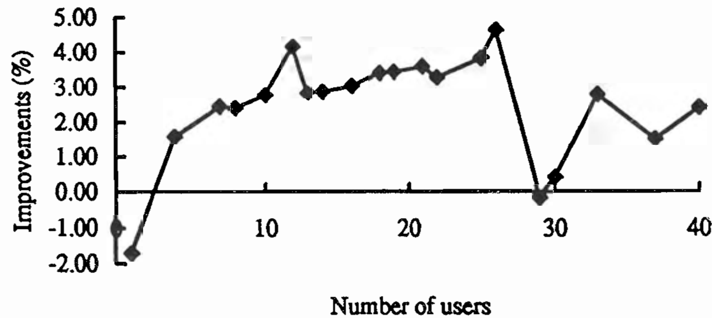



Figure 7. Improvements achieved in AIM benchmark. Compared with IPU policy, the performance improves by as much as 4.6%.

## 5. Related Works

UNIX systems have traditionally used a simple periodic update (PU) policy: once every 30 seconds, all dirty blocks in the file system's buffer cache are placed on the disk queue for updating [Bach 1986].

Carson and Setia showed that the PU policy actually performs worse in many cases than the write-through policy [Carson & Setia 1992]. Ruemmler and Wilkes also noted the write burst problem caused by periodic update policy [Ruemmler & Wilkes 1993].

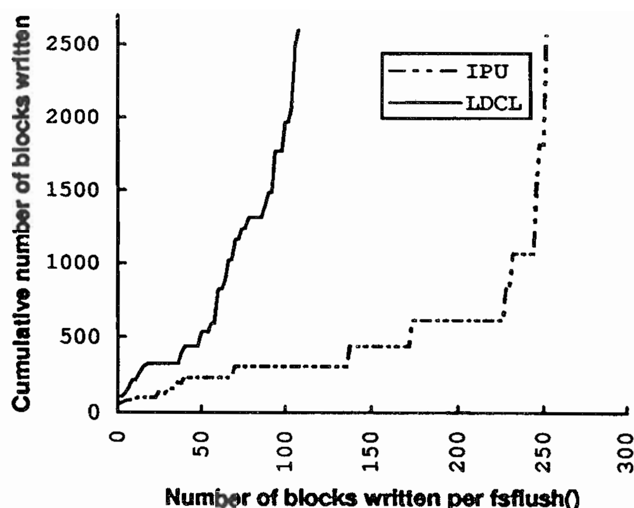



Figure 8. Burst-size distributions for writes traced during six runs of AIM III. The workload is 2–12 users and stepped by 2 users after each run.

Carson and Setia proposed several alternative write policies: WT, PURP, and IPU policy. They showed that IPU never gives worse mean read response time than WT or PU, although in some situations it may perform worse than PURP [Carson & Setia 1992].

Mogul used implementation to validate the results of Carson and Setia with an actual system. He found that combining delayed writes with IPU policy improves both the mean and variance. He also concluded that IPU policy depends on a relatively uniform distribution of file writes to achieve its more uniform distribution of disk writes, and higher performance [Mogul 1994].

Hac proposed dynamic update algorithms that choose when to schedule disk writes based on the system load and write activity [Hac 1991]. We share the same view, but she only proposed a conceptual model. Although his analysis proceeded with care, some important factors were overlooked. Besides, the advantages of dynamic policy are not visible in his performance analysis.

Other works also take the advantage of idle time to improve their performance. For example, in the AFRAID system, Savage and Wilkes alleviate the small update problem in a RAID 5 disk array by delaying the parity update to the next quiet period between bursts of client activity [Savage & Wilkes 1996]. By trading away a fraction of the reliability provided by disk array, it is possible to achieve performance that is almost as good as array of disks with no parity.

## 6. Conclusions

We have designed and implemented an intelligent update policy that smooths down the burstiness of input requests, and uses the I/O idle periods to accomplish opportunistic asynchronous write. The results of random read/write tests show improvements of as much as 40% in the variance, and 90% in the worst case read response time as compared with the IPU policy. Overall system performance is also improved according to AIM benchmark results.

There are some issues needing further study. For example, how to adjust the declustering threshold and the opportunistic asynchronous write control parameter more dynamically according to application burstiness. These studies can help us to control the mechanism more precisely, and construct a more effective update policy.

## References

1. M. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.
2. M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, Measurements of a Distributed File System, *Proceedings of 13th ACM Symposium on Operating System Principles*, pages 198–212, October 1991.
3. T. Blackwell, J. Harris, and M. Seltzer, Heuristic Cleaning Algorithms in Log-Structured File Systems, *Proceedings of the 1995 USENIX Technical Conference*, pages 277-288, January 1995.
4. A. L. Buck and R. A. Coyne, An Experimental Implementation of Draft POSIX Asynchronous I/O, *Proceedings of the Winter 1991 USENIX Conference*, pages 289–306, January 1991.
5. S. D. Carson and S. Setia. Analysis of the Periodic Update Write Policy For Disk Cache, *IEEE Transactions on Software Engineering*, Vol. 18, No. 1, pages 44–54, January 1992.
6. S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham, Transarc Corporation, The Episode File System, *Proceedings of the Winter 1992 USENIX Conference*, pages 43–60, January 1992.
7. L. C. Feng and R. C. Chang, Using Asynchronous Write on Metadata to Improve File System Performance, accepted by *The Journal of Systems and Software*, 1994.
8. G. R. Ganger and Y. N. Patt, The Process-Flow Model: Examining I/O Performance from the System's Point of View, *Proceedings of 1993 ACM SIGMETRICS*, pages 86-97, May 1993.
9. G. R. Ganger and Y. N. Patt, Metadata Update Performance in File System, *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 49–60, November, 1994.
10. A. Hac, Design Algorithms for Asynchronous Write Operations in Disk-Buffer-Cache Memory, *Journal of Systems and Software*, Vol. 16, No. 3, pages 243–253, November 1991.
11. J. H. Howard, M. L. Kazar, S. G. Nichols, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, Scale and Performance in a Distributed File System, *ACM Transaction on Computer Systems*, Vol. 6, No. 1, pages 51–81, February 1988.
12. R. Karedla, J. S. Loveffler, and B. G. Wherry, Caching Strategies to Improve Disk System Performance, *IEEE Computer*, Vol. 27, No. 3, pages 38–46, March 1994.
13. S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.
14. M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, A Fast File System for UNIX, *ACM Transactions on Computer Systems*, Vol. 2, No. 3, pages 181–197, August 1984.
15. L. W. Mcvoy and S. R. Kleiman, Extent-like Performance from a UNIX File System, *Proceedings of the Winter 1991 USENIX Conference*, pages 33–43, January 1991.

16. J. C. Mogul, A Better Update Policy, *Proceedings of the 1994 Summer USENIX Conference*, pages 99–111, June 1994.
17. K. Muller and J. Pasquale, A High Performance Multi-structured File System Design, *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 56–67, Oct. 1991.
18. J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, A Trace-Driven Analysis of the UNIX 4.2 BSD File System, *Proceedings of 10th ACM Symposium on Operating System Principles*, pages 15–23, December 1985.
19. J. K. Peacock, File System Multithreading in System V Release 4 MP, *Proceedings of the 1992 Summer USENIX Conference*, pages 19–29, June 1992.
20. P. J. Roy, UNIX File Access and Caching in a Multicomputer Environment, *Proceedings of the USENIX Mach III Symposium*, pages 21–37, April 1993.
21. C. Ruemmler and J. Wilkes, UNIX Disk Access Patterns, *Proceedings of the 1993 Winter USENIX Conference*, pages 405–420, January 1993.
22. S. Savage, and J. Wilkes, AFRAID—A Frequently Redundant Array of Independent Disks, *Proceedings of the 1996 USENIX Technical Conference*, pages 27–39, January 1996.
23. A. J. Smith, Disk-Cache Miss-Ratio Analysis and Design Considerations, *ACM Transactions on Computer Systems*, Vol. 3, No. 3, pages 161–203, August 1985.
24. C. Staelin, and H. Garcia-Molina, Smart File Systems, *Proceedings of the Winter 1991 USENIX Conference*, pages 45–51, January 1991.
25. A. S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1992, pages 175-179.
26. UNIX Software Operation, *UNIX System V Release 4 : System Administrator's Guide*, Prentice-Hall, Englewood Cliffs, NJ, 1990, pages 5:77–118.