Regular Paper

DiscNice: User-level Regulation of Disk Bandwidth

Hiroshi Yamada† and Kenji Kono†

A recent trend in computing has been to leverage dormant PC resources. To achieve this, background applications such as peer-to-peer applications and PC Grid run on ordinary users' PCs, sharing their computing resources. If not properly managed, the background applications obtrude on the PC user's active jobs. In particular, the contention over disk bandwidth severely degrades performance. In this paper, we present DiscNice, a novel scheme for disk bandwidth management that can host background applications unobtrusively. Its novelty lies in the fact that it throttles disk I/O completely at the user-level. The user-level approach is attractive for heterogeneous environments such as differently configured PCs over the world; portability is enhanced and deployment is easier in comparison with kernel-level approaches. Experimental results suggest that our prototype DiscNice running on Linux 2.4.27 incurs 12% or less overhead, and gracefully ensures the unobtrusiveness of background applications.

1. Introduction

A recent trend in computing has been to leverage dormant PC resources. To achieve this, background applications aggregate the resources of ordinary users' PCs to accomplish their tasks. Network applications such as peer-to-peer (P2P) applications and PC Grid are examples of background applications. Examples of P2P applications include KaZaA [1] and Gnutella [16]. They run on a huge collection of ordinary users' PCs to provide file sharing services. A feature of these applications is that they borrow user resources. In other words, the user lends resources such as the CPU, the memory, and the disk to these applications. A PC Grid such as SETI@home [13] is another example of background application. SETI@home uses the CPU time, the disk space, and the network I/O of its users' PCs to search for artificial radio signals coming from other stars. Housekeeping utilities such as backup utilities and virus scanners are other kinds of background applications. They are executed on ordinary users' PCs to keep them stable.

This new style of computing poses a significant problem for resource management. If not properly managed, background applications impede the execution of the PC user's active jobs, because commodity operating systems (OSes) cannot control the amount of resources allocated to the background applications. This problem is widely recognized [12], and has been tackled by many research projects [9],[11],[17]. These projects differed in the target resources they attempted to control and the layers (either at the user- or kernel-level) in which resource management is implemented. For example, the user-level scheduler that Newhouse *et al.* proposed [14] controls the CPU time at the user-level, whereas idletime scheduler [8] controls the network and disk I/O at the kernel-level.

To the authors' knowledge, however, no prior work has addressed the issue of controlling the disk I/O rate at the *user-level*. This is probably because disk I/O is not conceivable at the user-level. Disk I/O is not always accompanied by file I/O issued by processes, because the underlying OS caches and reads disk blocks in advance. Therefore, disk I/O cannot be directly controlled at the user-level. Despite the difficulty of user-level control over disk I/O, disk I/O still needs to be controlled at the user-level. Since recent computing environments naturally consist of heterogeneous hardware and OS platforms, it would be unrealistic to assume we could alter all OSes including proprietary ones like Windows Vista. Therefore, a user-level mechanism is required to enhance the portability and facilitate the deployment of such a mechanism.

This paper presents the design and implementation of *DiscNice*, a mechanism for controlling disk bandwidth at the user-level. To control disk I/O at the user-level, DiscNice infers what the internal behavior of the underlying OS is and predicts the disk I/O size that will be caused by file I/O. To infer internal kernel behavior, we extensively used a concept called *graybox* technology [2] and elaborated it

† Keio University

to predict the disk I/O behavior. In graybox technology, the underlying OS is treated as a graybox, which means that we could exploit 1) our knowledge of the OS, 2) the state information the OS exposes to us, and 3) how the OS reacts to various operations to predict the internal kernel behavior. By exploiting the graybox knowledge on Linux, we developed a graybox technique for predicting the disk I/O behavior. Our technique could also be applied to Windows Vista with minor modifications because it does not rely on a detailed knowledge of Linux.

Unlike conventional approaches which just stop the backgrounds or lower their priorities, DiscNice provides graceful disk I/O sharing between foregrounds and backgrounds. DiscNice enables the users to determine the amount of disk bandwidth assigned to the backgrounds. By adjusting the disk I/O rate for backgrounds, the users can regulate the degree of disk I/O contention. For example, a backup utility is assigned 1 MB/s of disk bandwidth and run with the foregrounds. Since the backup utility runs with at most 1 MB/s of disk bandwidth, the foregrounds can run with reduced disk I/O contention.

To determine how 'nice' DiscNice is, we implemented the prototype on Linux 2.4.27. The experimental results suggested that it could predict the disk I/O behavior accurately enough to control the disk I/O rate. DiscNice controlled the disk I/O rate successfully and prevented background applications from degrading the performance of the PC user's active jobs. In addition, the overhead incurred by using DiscNice was less than 12%.

The rest of the paper is organized as follows. Section 2 describes the necessity for the control of disk I/O and addresses its difficulties. Section 3 describes the key concept behind the graybox technology. Sections 4 and 5 describe the design and implementation of DiscNice. Section 6 presents the experimental results. Section 7 describes work related to ours. Section 8 concludes the paper.

2. Necessity and Difficulty

2.1 Disk I/O is Obtrusive.

Disk access is one of the heaviest tasks imposed on computer systems and has a significant impact on the performance of applications. A background application often requires disk access. For example, a backup utility accesses a lot of files and directories from local disks to preserve them on remote or other local storages. KaZaA [1] searches files requested by a user and stores them on the local disk. It is well known that the disk access seriously degrades the performance of PC user's active or foreground jobs.

Table 1 Performance degradation of `sequential` caused by disk I/O contention with `rsync`.

Benchmark	Execution time [sec]	Increasing rate
w/o `rsync` [sec]	4.18	—
w/- `rsync` [sec]	17.49	418%
w/- low priority `rsync`	17.26	413%

Table 2 Performance degradation of `tar` caused by disk I/O contention with `rsync`.

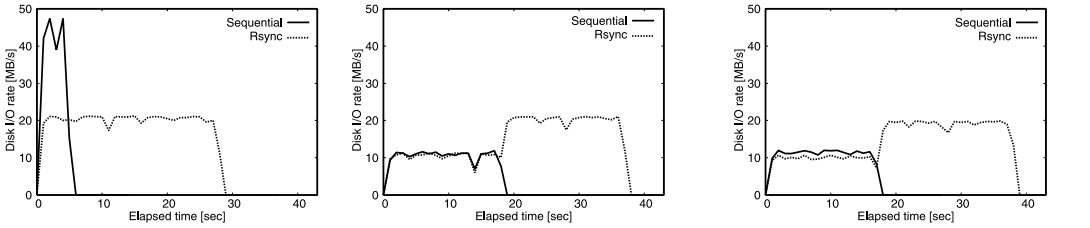
Benchmark	Execution time [sec]	Increasing rate
w/o `rsync` [sec]	4.47	—
w/- `rsync` [sec]	13.10	293%
w/- low priority `rsync`	12.82	287%

To assess the extent of degradation caused by background applications, we conducted an experiment on a 2.8 GHz Pentium4 PC with 1024 MB of memory and a SCSI HDD, running Linux 2.4.27. We prepared a background application `rsync` that backs up a 600 MB file to a remote storage and two foreground applications as follows.

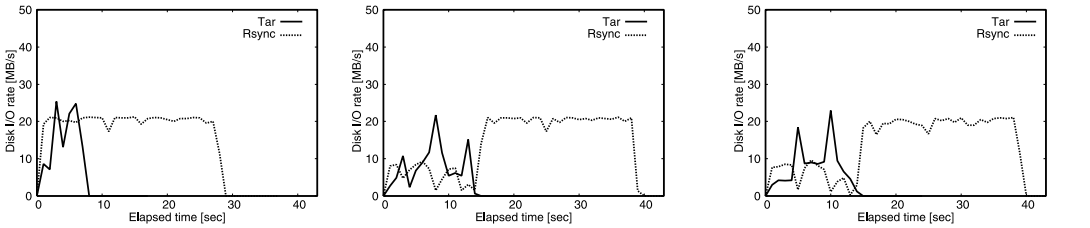
- `Sequential` : Reads a 200 MB file sequentially
- `Tar` : Unpacks a 50 MB archive file.

We measured the execution time for the two foreground applications and the resources usage in three situations. First, we executed each foreground application without the background application (`rsync`). They ran without resources contention. Second, we measured the usage and the time required to complete the foregrounds with the background application executed. Here, the foreground application competed with the background for disk bandwidth. In the last case, we executed the foregrounds with the background application whose CPU priority was lowered by setting its nice value to 19. To measure disk I/O rate per each process, we modified the Linux 2.4.27.

Table 1 and **Table 2** list the execution times of foreground applications in the three situations. The execution time of `sequential` with `rsync` is about 4.2 times longer than without `rsync`. The execution time of `tar` with `rsync` is about 2.9 times longer than without `rsync`. Even if the CPU priority of the background is lowered, the performance of the foregournds still remains low.

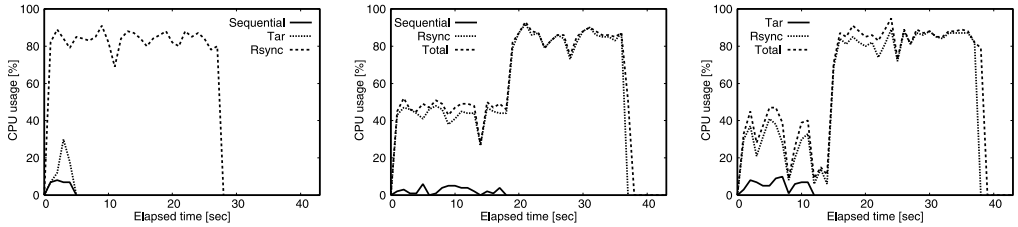


(a) Standalone (b) `Sequential` with `rsync` (c) `Sequential` with low priority `rsync`



(d) Standalone (e) `Tar` with `rsync` (f) `Tar` with low priority `rsync`

Fig. 1 Disk bandwidth usage.



(a) Standalone (b) `Sequential` with `rsync` (c) `Tar` with `rsync`

Fig. 2 CPU usage.

Figure 1 shows disk bandwidth usage in the experiment. We plotted a mean value within one second as disk I/O rate, based on a plotting scheme used by the other paper [18]. At first glance, the execution time of `tar` is different between Table 2 and Fig. 1. This is caused by Delayed Write, one of the OS features to improve system performance, as described in Section 2.2. Therefore, disk I/O requests are issued after `tar` finished its execution.

We can see that disk I/O contention occurs when `sequential` and `tar` run concurrently with `rsync`. When executed concurrently, disk I/O rates of `sequential` and `rsync` are about 11 MB/s and 10 MB/s respectively, although they are about 47 MB/s and 20 MB/s when executed in standalone. After `sequential` finished executing, disk I/O rate of `rsync` goes back to 20 MB/s as in standalone because disk I/O contention finished. Even if we lower CPU priority of `rsync`, the disk I/O contention is hardly resolved. The situation similar to that described above occurs in the case the foreground application is `tar`.

We show CPU usage of the experiment in **Fig. 2** to confirm that the degradation of the disk bandwidth usage is not caused by CPU contention. Figure 2 suggests that CPU contention did not occur. Even when the foreground application is executed with `rsync`, CPU usage of the system is at most 51%. In addition, Fig. 2 reveals that the CPU usage is lowered when the foreground is executed with `rsync`. Since disk I/O contention delays seeking data on disk, every application requesting data on the disk cannot make progress and thus CPU usage is lowered.

Figure 3 shows the memory usage. From this figure, the memory was not exhausted in this experiment. Even if the foreground is executed with the background application, the memory usage is less than the available memory.

2.2 Difficulty

Disk I/O should be regulated because the disk accesses of background applications de-

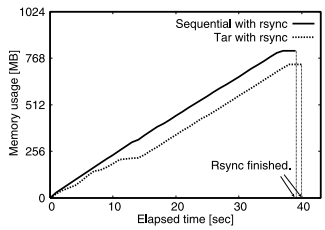


Fig. 3 Memory usage.

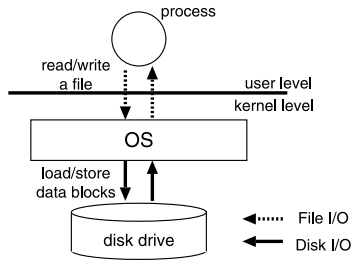


Fig. 4 Difference between file I/O and disk I/O.

grade the performance of foreground jobs. We propose DiscNice, a novel scheme of managing disk I/O at the *user-level*. To control disk I/O, we need to understand the difference between *file I/O* and *disk I/O*.

Figure 4 outlines the difference between file I/O and disk I/O. File I/O refers to the read/write requests issued by a process. Note that the file I/O issued by the processes can be easily and completely monitored at the user-level. Disk I/O, on the other hand, refers to the load/store operations performed by the OS to physically access the disk drive. To control the contention over disk bandwidth, we must regulate disk I/O, not file I/O, because file I/O does not always cause disk I/O. For example, reading a file does not always load data from the disk drive. Even if data is actually loaded, the disk I/O size is not always the same as that for file I/O. These are caused by the underlying OS for the following reasons.

- **Disk Cache:** Reading a file within the disk cache does not incur disk I/O.
- **Block-Based Access:** Disk access is done in units called blocks. Thus, the disk I/O size is not always equal to the file I/O size.
- **Read-Ahead:** The underlying OS reads one or more blocks of a file in advance. Thus, the disk I/O size is often larger than the file I/O size.
- **Delayed Write:** The underlying OS delays writing back dirty blocks to the disk until the dirty buffer becomes full. This delay makes disk I/O asynchronous with file I/O for writes. Since the asynchronous write combines multiple writes on the same file location, the disk I/O size is not always equal to the file I/O size. This is described in more detail in Section 4.4.

3. Graybox Technology

To control the contention over disk bandwidth at the user-level, DiscNice monitors the file I/O performed by target applications and predicts the disk I/O behavior in the OS kernel. As described in Section 2.2, the decision of whether to issue disk I/O or not depends on OS behavior. Therefore, it is difficult to precisely predict disk I/O at the user-level unless the OS provides information on internal behavior related to disk I/O.

Unfortunately, the popular OSes do not expose these pieces of information. Thus, a mechanism that will predict disk I/O is needed. In this paper, we predict the disk I/O by applying the *graybox* technology proposed by Arpaci-Dusseau, et al. [2]. This enables us to predict the internal state of the OS at the user-level. When treating the OS as a graybox, we do not change the OS source code but exploit the general characteristics of the algorithms employed by the OS. By exploiting this knowledge, we can predict the internal state of the OS even if there is no interface to obtain this state. More specifically, we could exploit 1) our knowledge of the OS, 2) the state information the OS exposes to the users, and 3) how the OS reacts to various operations. Linux could easily be treated as a graybox, because its source code is open and the `proc` file system provides some of Linux's internal states.

Arpaci-Dusseau, et al. [2] applied graybox technology to predict whether there was a file in the disk cache. This prediction was used to improve the performance of `grep`. If multiple files are to be processed, the modified `grep` predicts which file is already in the disk cache, and reorders the file operations so that the file already in the cache can be accessed first. By doing this, the execution time of the modified `grep` is about three times faster than the default `grep`.

Unfortunately, their technique cannot accurately predict disk I/O behavior. Disk I/O behavior depends on not only the state of the disk cache but also other internal states such as file read-ahead and block-based access. We extended their technique to predict disk I/O be-

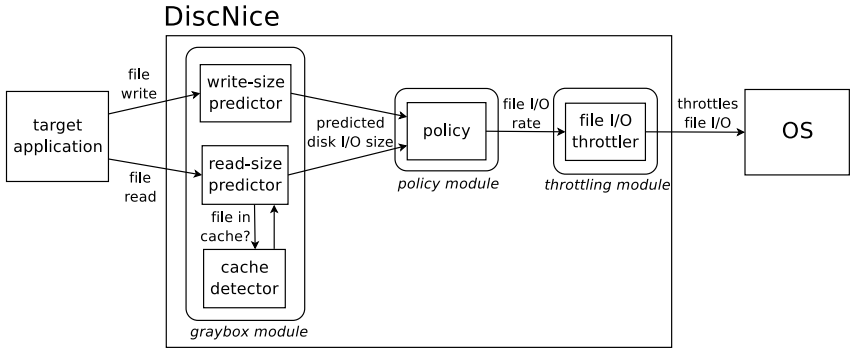


Fig. 5 Control flow in DiscNice.

havior.

4. User-level Disk Bandwidth Control

To regulate disk I/O, DiscNice throttles the file I/O performed by target applications. As described in Section 2.2, the file I/O is not always accompanied by disk I/O. To determine whether to throttle the file I/O, DiscNice predicts the size of disk I/O that will be caused by the file I/O. If disk I/O is not incurred, the disk I/O size is zero. Below, we summarize how DiscNice handles the following operations.

- **Disk Cache:** DiscNice must predict the presence of a file within the cache to avoid throttling the file I/O that hits the cache.
- **Block-Based Access:** To predict the disk I/O size, DiscNice rounds up the file I/O size to the multiples of the block size.
- **Read-Ahead:** DiscNice emulates the read-ahead algorithm employed by the underlying OS.
- **Delayed Write:** Disk I/O for writes is uncontrollable at the user-level because it is done asynchronously with file I/O requests. To keep the average rate of disk I/O lower than a threshold, DiscNice delays write requests for a while when a dirty buffer is flushed.

4.1 System Design

In the design of DiscNice, the throttling policy is separated from the throttling mechanism. DiscNice has three modules; the *graybox* module, the *policy* module and the *throttling* module. **Figure 5** shows the control flow in DiscNice. The graybox module predicts the disk I/O size when a target application performs file I/O. The policy module determines whether to throttle the file I/O based on the disk I/O size predicted by the graybox module. Then, the policy module calculates the rate of file I/O to throttle the disk I/O. The throttling module throttles the file I/O. This design allows us to change the throttling policy without modifying the mechanism. By changing the policy, the user can adjust the degree of disk bandwidth contention.

The graybox module consists of three submodules, i.e., *cache detector*, *read-size predictor*, and *write-size predictor*. The cache detector predicts whether there is a file within the disk cache. When a file is read, the read-size predictor predicts the actual disk I/O size by considering block-based accesses and read-ahead in the underlying OS. When a file is written on, the write-size predictor predicts the actual disk I/O size by considering block-based accesses and delayed write.

4.2 Cache Detector

The cache detector predicts whether a file to be read is in the disk cache. The algorithm used for the cache detector is the same as the one used in the Arpaci-Dusseau's graybox technique except for a minor tuning. The cache detector measures the time required to probe a file (i.e., read a single byte from the file). If the probe returns quickly, it judges that the file is within the disk cache. If the probe returns slowly, it considers the file is not in the cache.

When a target application reads a file, the M-bytes region from the reading point (file pointer, fp) is examined to detect which parts of the region are within the disk cache. We divide the M-bytes region into smaller m-bytes regions. We probe $\lfloor M/m \rfloor$ points in the M-bytes region; i.e., $fp+m$, $fp+2m$, ... , $fp+\lfloor M/m \rfloor \cdot m$. If the probe into $fp + k \cdot m$ returns quickly, the m-bytes region from $fp + (k-1) \cdot m$ is regarded as being in the disk cache. If the probe returns slowly, the region is not considered to be in the cache. The following reads do not al-

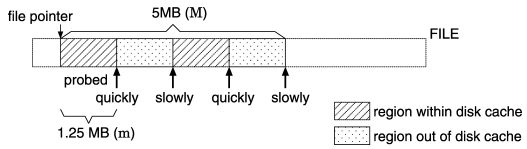


Fig. 6 An example of cache detection.

ways invoke the cache detector. If the current file pointer is within the M-bytes region that has already been examined, the cache detector is not invoked. The results from the previous probes are reused. Otherwise, the cache detector is invoked as previously described.

For example, imagine that a target application issues `read(fd, buf, 4096)`. Let M and m be 5 MB and 1.25 MB respectively, as outlined in **Fig. 6**. If the probes into the points at 1.25 MB and 3.75 MB return quickly, the cache detector considers the region from the reading point to the 1.25 MB point and the region from the 2.5 MB point to the 3.75 MB point to be within the disk cache. After that, if the target application issues `read(fd, buf, 8192)`, the cache detector is not invoked because the current file pointer is within the M-bytes region already examined.

M must be selected carefully. If M is too large, the conditions for the M-bytes region may change until the current file pointer goes beyond the M-bytes region, and this lowers the accuracy of the cache detection. If M is too small, the cache detector always considers the entire region is within the cache because the underlying OS reads the file in advance and the region around the file pointer is usually in the cache. In the current implementation, M is set to 5 MB.

4.3 Read-Size Predictor

The read-size predictor predicts the disk I/O size for reads. When the target application issues `read(fd, buf, n)`, the read-size predictor invokes the cache detector. If the file to be read is in the cache, the predicted disk I/O size is zero. If the read file is not in the cache, the read-size predictor rounds up `n` to multiples of the block size. Then, it adds the read-ahead size to the rounded `n` by emulating the read-ahead behavior of the underlying OS.

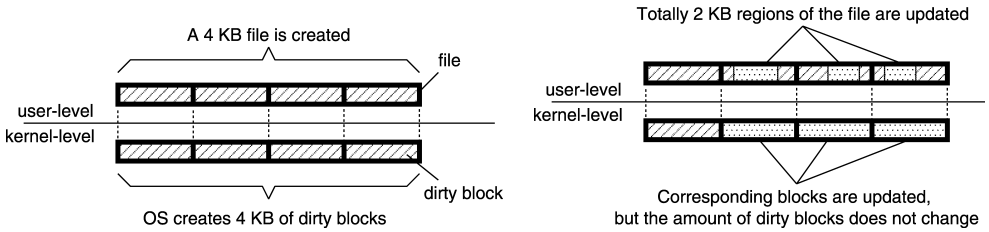
To emulate read-ahead, we do not need the full knowledge of the read-ahead algorithm used by the OS. It is unnecessary to emulate the read-ahead algorithm accurately although our method involves some errors. In most cases, rough emulation would be sufficient to control disk I/O, as our experiments suggest; the error with read prediction by DiscNice is less than 5%, even though it does not fully emulate Linux's read-ahead.

4.4 Write-Size Predictor

To predict the disk I/O size for writes, we must carefully consider not only block-based access but also delayed write. Delayed write is an OS behavior that delays writing back dirty blocks to the disk until the dirty buffer becomes full. Since it combines multiple writes on the same file location, the disk I/O size is not always equal to the file I/O size. For example, imagine that a process opens a new file and writes 4 KB data on the file, then updates totally a 2 KB region of it. When the 4 KB data is written on that file, the OS creates 4 KB dirty blocks in memory (**Fig. 7** (a)). When the file is updated, the OS writes the updates on the dirty blocks (Fig. 7 (b)). In this case, although the total file I/O size is 6 $(= 4 + 2)$ KB, the actual disk I/O size is 4 KB; the updates do not cause disk I/O because the OS does not create any new dirty blocks. In the same example, if the file is deleted instead of being updated, the total file I/O size is 4 KB but the actual disk I/O size is zero; the OS removes the dirty blocks to avoid unnecessary disk writes.

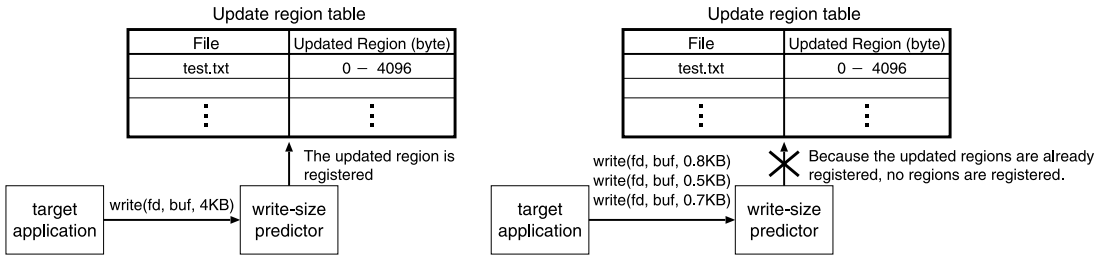
To deal with the difference generated by delayed write, we prepared an *update region table* which retains updated regions of files until the dirty buffer is written back to the disk. When a target application issues `write(fd, buf, n)`, the write-size predictor calculates the updated region of the file on which the application writes data; the write-size predictor regards $[\lfloor \mathtt{fp}/b \rfloor \cdot b, \lceil (\mathtt{fp} + \mathtt{n})/b \rceil \cdot b)$ as being updated, where b denotes the block size. Then, the write-size predictor refers to the update region table. If the calculated region is registered in the table, the write-size predictor considers that the `write()` will not incur disk accesses. Otherwise, the write-size predictor registers the region in the table. If a file is unlinked, the registered regions of the file are deleted. When the underlying OS writes the dirty blocks back to the disk, the write-size predictor calculates the total size of the updated regions by looking up the table and regards the calculated size as the disk I/O size for writes. After that, the write-size predictor notifies the policy module of the size and clears the update region table.

In the above example (shown in Fig. 7), when a file is opened and written, the size predic-



(a) When a file is created (b) When the file is updated

Fig. 7 OS behavior in file creation and updates.



(a) When a file is created (b) When the file is updated

Fig. 8 Write-size predictor behavior in file creation and updates.

tor registers the 4 KB region in the update region table (**Fig. 8** (a)). When the file is updated, the update region table is not changed because the updated regions are already registered (Fig. 8 (b)). When the dirty buffer is flushed, the write-size predictor sends "4 KB" (the total size of the registered regions) to the policy module and clears the update region table.

5. Implementation

We developed the prototype DiscNice on Linux 2.4.27. The prototype is composed of the `libDiscNice` library, the `write-adjuster` process, and the `policyd` process. **Figure 9** outlines DiscNice's architecture. `LibDiscNice` is a dynamic library linked with a target application. It contains the cache detector, the read-size predictor, and the throttling module. It monitors the file I/O performed by the target application. When a target application requests to read files, `libDiscNice` predicts the disk read size by calling the read-size predictor and the cache detector. When a target application requests to write on or delete files, `libDiscNice` notifies the `write-adjuster` of the requests. The `write-adjuster` is a process that implements write-size predictor. Write-size predictor must collect write requests of target applications to predict the disk I/O size with the update region table. Write-size predictor is separated from `libDiscNice` because this design makes it easier to manage the update region table. `LibDiscNice` and `Write-adjuster` send the predicted disk I/O size to the `policyd` process. After receiving requests to write on or delete files, write-size predictor is invoked. Then, the `policyd` process determines the file I/O rate based on a given policy. Finally, the throttling module in the `libDiscNice` requests the file I/O rate from the `policyd` and throttles the requested file I/O on the rate as directed by the `policyd` process.

5.1 LibDiscNice

`LibDiscNice` hooks all file I/O requests. It overrides the standard functions in `libc` related to file I/O. In the current implementation, `read()`, `write()`, and `unlink()` are overridden. To override these functions, we used the facility of library preload that forces a specific library to be linked before all other libraries. In Linux, we can specify the preloaded library with the environment variable `LD_PRELOAD`.

After hooking the request to read files, `libDiscNice` predicts and notifies `policyd` process of the disk read size. When write or delete requests are hooked, `libDiscNice` notifies `write-adjuster` of the requests. `LibDiscNice` communicates with the `policyd` and `write-adjuster` through a TCP/IP connection. When the predicted disk I/O size is zero, it is not sent to `policyd`, to avoid unnec-

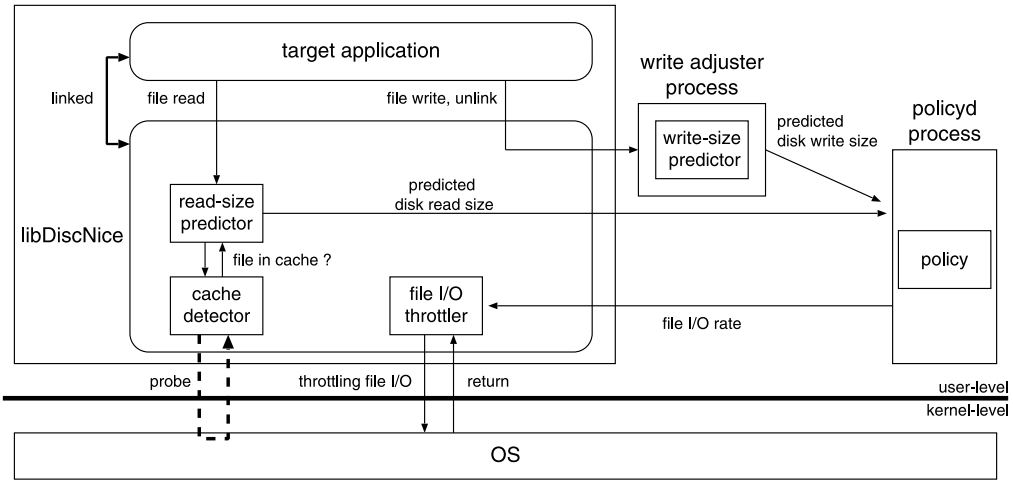


Fig. 9 DiscNice architecture.

essary interprocess communications.

When a child process is spawned by a target application, it is automatically throttled by DiscNice because it is also linked with `libDiscNice`. In the current policy, the disk I/O issued by the child process is charged to the parent process. If necessary, the child process can be charged separately from the parent process.

5.2 `Write-adjuster`

When a target application invokes `write(fd, buf, n)`, `libDiscNice` sends the current position of the file pointer and `n` to `write-adjuster`. Then, `write-adjuster` invokes the write-size predictor. The write-size predictor checks whether the updated region is registered in the update region table, and registers it if necessary.

The update region table is a hash table of the `wpred_st` structure, which is looked up with an inode-number. The `wpred_st` structure has the inode-number of a file and an array which retains the update regions of the file.

```
struct wrepd_st {
      /* file's inode-number */
      ino_t   ino;

      /* updated region */
      boolean wpos[MSIZE];
};
```

The array `wpos` retains the updated ranges of the file in 4 KB units because Linux 2.4.27 accesses files in the disk in 4 KB units. If a value of `wpos[i]` is true, the region $[i \cdot 4\,KB, (i+1) \cdot 4\,KB)$ is regarded as being updated. For example, when a target application opens a new file and writes 100 KB data on it from the beginning, the write-size predictor creates a new entry in the update region table. Then, it pushes trues in the indices from 0 to 24 ($= 100\,KB/4\,KB - 1$) of the `wpos` array. If the file is unlinked, the write-size predictor deletes the entry associated with it. To predict the disk I/O size for writes, the wsp counts the number of trues in the update region table. In the prototype, the `wpos` is implemented as a simple array. To deal with huge files, the `wpos` should be implemented as a tree structure.

When the OS writes dirty blocks back to the disk, the write-size predictor sends the predicted size to the `policyd` process and then clears the update region table. The write-size predictor monitors the `proc` file system to know when the dirty buffer is flushed. The `proc` exposes the number of written disk blocks in the whole system. By monitoring the change of this value, the write-size predictor detects the time of flushing of the dirty buffer. We believe that our scheme is applicable to different OSes because modern OSes expose the internal states. For example, Windows supports System Information Functions, Solaris supports Dtrace [4] and so on.

To reduce the prediction overhead, we reduced the number of accesses to the `proc` file system. It is expensive to access the `proc` file system every time `write()` is called. In the prototype, the write-size predictor checks the `proc` file system every 500 ms. As shown in Section 6.1 and 6.3, this optimization does not incur serious loss of prediction accuracy.

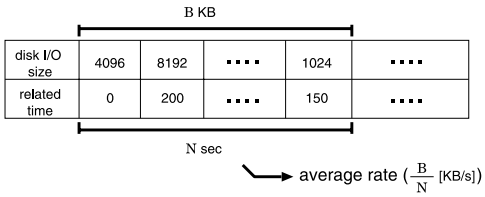


Fig. 10 Rate-window.

5.3 `Policyd` Process and the Current Policy

The `policyd` process receives the predicted disk I/O size and then invokes function `policy()` which implements the throttling policy. The signature of `policy()` is policy(pid_t pid, size_t predicted). The `policy()` takes the process ID and the predicted disk I/O size. It returns the file I/O rate by which the process specified by `pid` should perform the file I/O.

The current policy employs rate-windows [18]. It restricts the disk I/O rate performed by a target application below a user-defined *threshold*. To regulate disk I/O rate, we used a sliding window of recent disk I/O requests to compute the average rate for a target application. The target process is put to sleep when it requires disk I/O that would cause its disk I/O rate to exceed the threshold.

When a target application requires disk I/O, the current policy records the following information; the predicted disk I/O size and the time elapsed from the last disk I/O. The policy calculates the disk I/O rate during the last N seconds from these pieces of information (see **Fig. 10**). Let the threshold be R KB/s, and assume that the disk I/O size during the last N seconds is B KB. If $B/N \leq R$, the requested disk I/O would not exceed R. Therefore, `policy()` directs the throttling module to execute the file I/O. Otherwise, since the requested disk I/O would exceed R, `policy()` sleeps for $B/R - N$ time and then directs the throttling module to execute the file I/O. In the prototype, N is set to 1.

Note that B may become large when the dirty buffer is flushed because the underlying OS writes back dirty blocks to the disk in batches. As a result, B/N may exceed R just after the dirty buffer is flushed. In the current policy, DiscNice delays file I/O requests until B/N becomes less than R. By inserting the delay, DiscNice limits the *average* disk I/O rate to less than R.

We reduced the number of `libDiscNice`'s communications with the `policyd` to reduce the overhead. It is wasteful for `libDiscNice` to obtain a file I/O delay time from the `policyd` every time `write()` or `unlink()` is called, because OS writes blocks to the disk in batches. The `libDiscNice` requests the time per 300 ms. This does not seriously affect prediction accuracy, as is shown in Section 6.1 and 6.3.

5.4 Read-Size Predictor

There are several things to be noted in implementing the read-size predictor. In Linux 2.4.27, the file in the disk drive is accessed in the unit of page size (4 KB), even though the disk block size is 1 KB. Therefore, DiscNice regards the block size as 4 KB.

To emulate read-ahead in Linux, we examined the source code of Linux 2.4.27. Linux 2.4.27 employs two modes for file read-ahead: synchronous and asynchronous [3]. Linux starts reading a file in synchronous mode, in which the read-ahead size is fixed at 4 KB. If the next read is sequential, Linux switches to asynchronous mode. In asynchronous mode, the read-ahead size increases while the file is read sequentially. Otherwise, Linux remains in synchronous mode.

DiscNice only emulates the synchronous mode. Emulating the asynchronous mode would increase the accuracy of disk I/O prediction. However, when the asynchronous mode is used with DiscNice, the read-ahead size will not increase. Since the probes done by the cache detector are regarded as random access, the read-ahead size is reset to 4 KB periodically. Thus, disk I/O prediction is still sufficiently accurate even if the asynchronous mode is not emulated. In fact, the experimental results in Section 6 demonstrate that precision is satisfactory. Furthermore, the overhead caused by nullifying the asynchronous mode is not large in actual applications.

5.5 Discussion

Linux has disk I/O not induced by file I/O. The `mmap()` system call maps files to memory. When the mapped memory is accessed for the first time, Linux loads the accessed page from the disk into the memory. Therefore, the current DiscNice cannot detect disk I/O that have originated from `mmap()`. DiscNice can be extended to detect this kind of disk I/O; It protects the mapped memory with the `mprotect()` system call so that the `libDiscNice` can be notified of access to the memory. Since a page fault only occurs when a memory page has been

Table 3 Experimental environment.

CPU	Pentium 4 2.8 GHz
Memory	1,024 MB
HDD	7200 rpm, SCSI
OS	Linux 2.4.27

accessed for the first time, the overhead would not be that large.

There is one more thing to be noted. Throttling malicious applications is beyond the scope of this paper. The current implementation of DiscNice assumes that background applications are trusted. Therefore, a malicious application can bypass the I/O control with DiscNice. For example, a malicious application could disable the `libDiscNice` by calling `read()`, `write()` and `unlink()` directly. Extending DiscNice to defend against these malicious applications is an interesting research topic that bears further investigations.

6. Experiments

We conducted experiments to find out how effective DiscNice is. The experimental environment is summarized in **Table 3**.

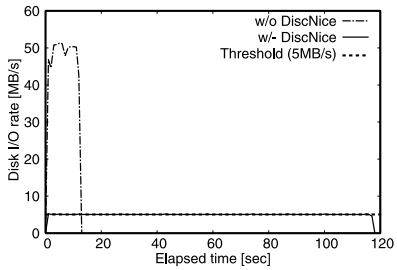
6.1 Micro-benchmark

To demonstrate how precise DiscNice's prediction is, we compared the actual disk I/O size to the predicted size. Linux 2.4.27 was modified to record the total disk I/O size per process. We prepared the following benchmark programs.

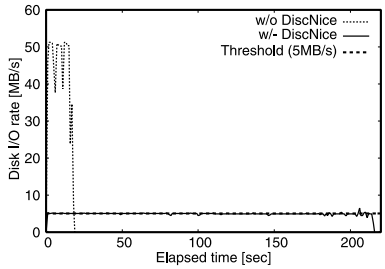
- `Sequential:` Reads a 600 MB file sequentially.
- `Random:` Reads a 600 MB file randomly 10 times.
- `Stride:` A single byte in a 600 MB file is read every 12 KB.
- `Create:` Creates a 400 MB file and fills it with the character 'a'.
- `Region-Write:` Write 10 KB data in the same region of a 400 MB file 10,000 times.
- `Create-Delete:` Creates and unlinks a 10 KB file 10,000 times.

Table 4 compares the real disk I/O size to the predicted disk I/O size. In all cases, the error margin is less than 1%.

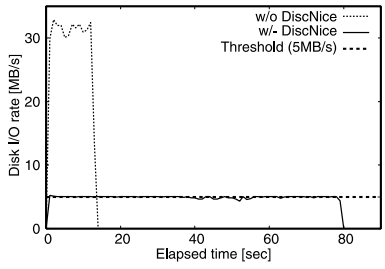
To find out whether DiscNice could regulate disk I/O, we measured the disk I/O rate for micro-benchmark programs except for benchmarks which did not almost issue disk I/O, `Region-Write` and `Create-Delete`. The threshold was set to 5 MB/s. For comparison, we also measured the disk I/O rate without running DiscNice. **Figure 11** plots the disk I/O rates for the benchmarks. The x-axis is the elapsed time, and the y-axis is the disk I/O rate. The solid lines plot the disk I/O rates controlled by DiscNice. The dotted lines plot the rates not controlled by DiscNice. In the graph for `create`, the disk I/O rate is averaged over intervals when the dirty buffer was flushed, because the disk write was batched. From Fig. 11, we can see that DiscNice successfully throttles disk I/O.



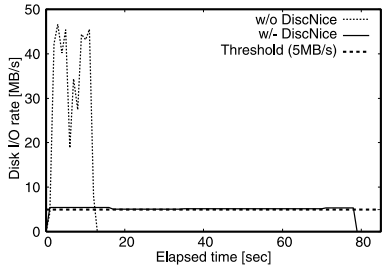
(a) `Sequential`



(b) `Random`



(c) `Stride`



(d) `Create`

Fig. 11 Results of throttling micro-benchmark programs.

Table 4 Accuracy of disk I/O prediction (micro-benchmark).

Benchmark	`Sequential`	`Stride`	`Random`	`Create`	`Region-Write`	`Create-Delete`
Actual disk I/O [byte]	615,022,592	411,250,688	491,073,536	409,612,288	24,576	0
Predicted disk I/O [byte]	615,006,208	410,173,440	492,625,920	409,600,000	24,576	0
Error margin	−0.00%	−0.26%	0.32%	−0.03%	0%	0%

Table 5 DiscNice configuration.

Configration Name	Cache detector	Read-size predictor	Write-size predictor
`All-off`	off	off	off
`CacheD`	on	off	off
`ReadP`	off	on	off
`CacheD&readP`	on	on	off
`All-on`	on	on	on

We can also say that the cache detector has a small side effect on disk I/O. The cache detector causes the extra disk I/O to predict whether a file to be read is in the disk cache. `Stride` is the worst case in our experiment because it performs a lot of small reads but DiscNice successfully throttles the disk I/O rate (Fig. 11 (c)).

6.2 Effectiveness of DiscNice Modules

To confirm each module of DiscNice contributes to controlling disk I/O rate, we prepared the benchmark consisting of the following four phases.

(1) Random-Read phase: Reads a 200 MB file randomly 10 times.
(2) Stride-Read phase: A single byte in a 200 MB file is read every 12 KB.
(3) Region-Write phase: Writes 10 KB data in the same region of a file 10,000 times.
(4) Create-Delete phase: Creates and unlinks a 10 KB file 10,000 times.

This benchmark was executed with DiscNice in the five configurations summarized in **Table 5**. Each configuration turned on and off some modules of DiscNice (cache detector, read-size predictor and write-size predictor).

The results are shown in **Fig. 12**. The threshold was set to 5 MB/s. In `all-off` which corresponds to the control of only the file I/O rate (Fig. 12 (a)), disk I/O is throttled excessively in the random-read phase because DiscNice throttles the file access for data in the disk-cache. In addition, it cannot throttle disk I/O derived from read-ahead and block-based access. As a result, disk I/O rate exceeds the threshold (5 MB/s) in the stride-read phase.

When the cache detector is turned on, DiscNice successfully avoids throttling file I/O that accesses data in the disk cache (Fig. 12 (b)). But the disk I/O derived from read-ahead and block-based access is not controlled well; the disk I/O rate exceeds the threshold in the stride-read phase.

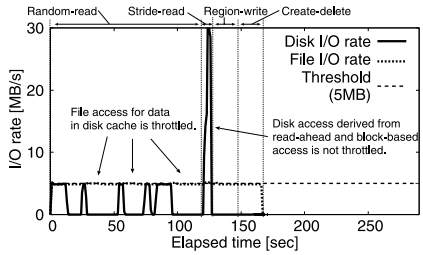
When the read-size predictor is turned on, DiscNice successfully throttles disk I/O derived from read-ahead and block-based access, but regulates excessively the access in the random-read phase (Fig. 12 (c)).

When the cache-detector and the read-size predictor are turned on, DiscNice controls disk I/O rate successfully in the stride-read phase (Fig. 12 (d)). DiscNice also unnecessarily throttles file write requests that will not cause disk I/O in region-write and create-delete phases.

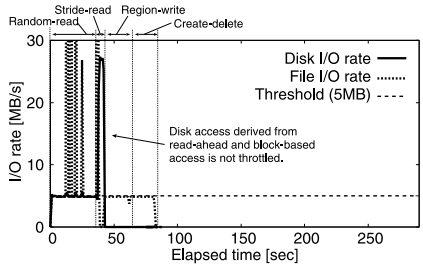
From Fig. 12 (e), we can see that `all-on` successfully throttles disk I/O rate without largely exceeding the threshold. This means that DiscNice with `all-on` avoids throttling file I/O which does not accompany disk I/O in the random-read, region-write and create-delete phases. Thus, the execution time of `all-on` is the shortest in all the configurations.

We also observed how each module of DiscNice affects the performance of a foreground application. We prepared a foreground application, `sequential`, which sequentially reads five 600 MB files. As a background application, we used a benchmark called `bench` similar to the one described above; we swapped the stride phase for the random phase in the above benchmark to explicitly show effectiveness of our modules. We concurrently executed the foreground application with the background controlled by DiscNice, and measured execution times. Note that the performance of `sequential` is degraded due to disk I/O contention with `bench`.

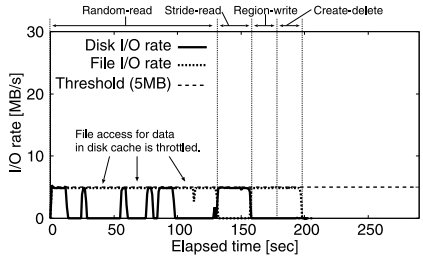
Table 6 lists the results. The threshold is set to 3 MB/s. At the first glance, the readers might consider that `all-off` and `readP` are better than `all-on` since the execution times of `sequential` are about 1% shorter than `all-on`. This is because DiscNice excessively throttles file I/O rate for `bench`. The excessive throttling of file I/O slightly shortens the execu-



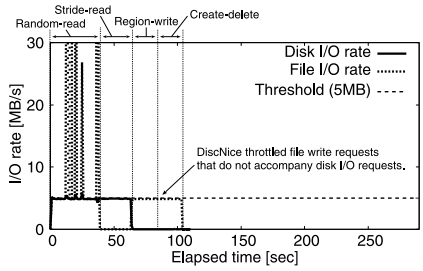
(a) `All-off`



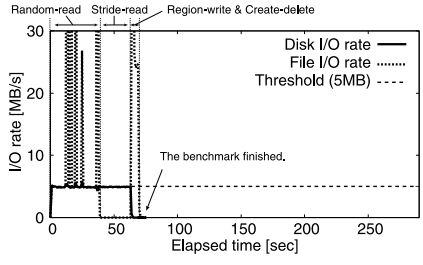
(b) `CacheD`



(c) `ReadP`



(d) `CacheD&readP`



(e) `All-on`

Fig. 12 Results of throttling micro-benchmark programs.

Table 6 Comparison of execution times in different DiscNice configurations.

Configuration	`Sequential` [sec]	`Bench` [sec]
`All-off`	371.47	601.49
`CacheD`	386.13	484.45
`ReadP`	371.90	629.12
`CacheD&readP`	375.26	448.40
`All-on`	375.32	414.57

Table 7 Accuracy of disk I/O prediction (macro-benchmark).

Benchmark	`Tar`	`Make`	`Grep`
Actual disk I/O [byte]	435,482,624	48,271,360	215,965,696
Predicted disk I/O [byte]	430,954,752	46,735,360	206,376,960
Error margin	−1.0%	−3.2%	−4.4%

tion time of the foreground application but the performance of the background application is severely degraded in `all-off` and `readP`; the execution times of `bench` in `all-on` and `readP` are at worst 1.5 times longer than the other configurations. On the other hand, the execution time of the foreground is slightly increased in `all-on` but the background finishes the execution fastest.

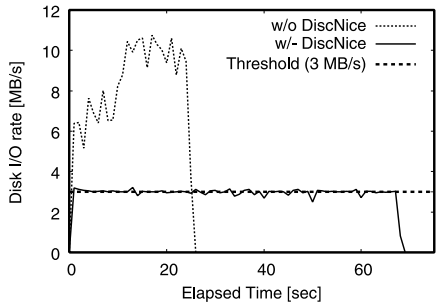
6.3 Macro-benchmark

To find out how capable DiscNice is in actual applications, we prepared the following benchmark programs.

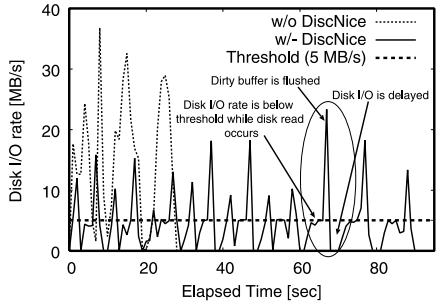
- `tar`: Unpacks a 200 MB archive file.
- `make`: Compiles Apache [21] 2.0.52. The total size of the source code is about 8 MB.
- `grep`: Searches for lines containing 'epoch' in the source code and documents for Linux kernel 2.4.27. The total file size is about 200 MB.

Table 7 compares the actual disk I/O size and the predicted disk I/O size. The prediction errors are less than 5% in the benchmarks. The error is slightly larger in `grep`. This prediction error is caused by reading directories since the current DiscNice does not hook a system call for reading these.

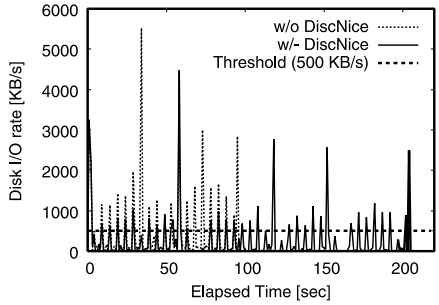
To find out whether DiscNice could regulate disk I/O in real applications, we measured the disk I/O rate for these benchmarks. The threshold was 5 MB/s for `tar`, 500 KB/s for `make`, and 3 MB/s for `grep`. **Figure 13** shows the results. The x-axis represents the elapsed time, and the y-axis is the disk I/O rate. The solid lines plot the disk I/O rates controlled by DiscNice. The dotted lines plot the rates not controlled by DiscNice. From Fig. 13, we



(a) `Grep` (Disk I/O threshold is set to 3 MB/s)



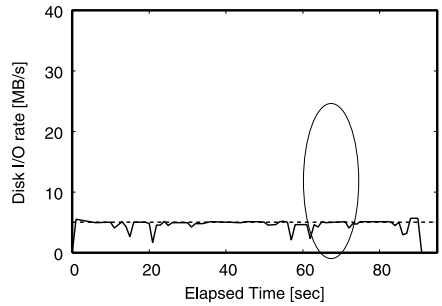
(b) `Tar` (Disk I/O threshold is set to 5 MB/s)



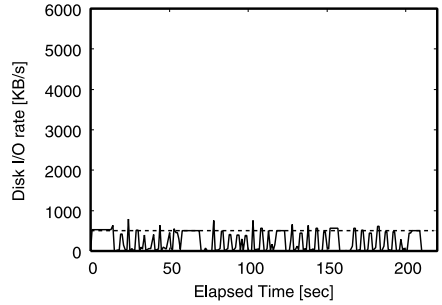
(c) `Make` (Disk I/O threshold is set to 500 KB/s)

Fig. 13 Results of throttling macro-benchmark programs.

can see that DiscNice works well for `grep` because it is the read-only benchmark. In the other benchmarks that involved write operations, there are some spikes that exceed the threshold lines. These spikes occur when the dirty buffer is flushed. For example, look at the circled region in Fig. 13(b). The disk I/O rate is below the threshold for a while. When the dirty buffer is flushed, it exceeds the threshold temporarily. After that, disk I/O is not issued, since DiscNice delays file I/O requests until the average disk I/O rate becomes lower than the threshold. **Figure 14** shows the average disk I/O rate over intervals between the buffer flush and the following disk I/O. The circled region in Fig. 14 (a) corresponds to the



(a) `Tar`



(b) `Make`

Fig. 14 Average disk I/O rate of `tar` and `make` with DiscNice.

Table 8 Overhead incurred by DiscNice.

Benchmark	Tar	Make	Grep
w/o DiscNice [sec]	29.87	97.13	27.87
w/- DiscNice [sec]	30.78	108.55	27.98
Overhead	3.05%	11.76%	0.40%

one in Fig. 13 (b). In Fig. 14, we can see that the average disk I/O rates in `tar` and `make` are each below the threshold. This suggests that DiscNice successfully throttles the disk I/O.

6.4 Overhead

To measure the overhead incurred by DiscNice, we compared the execution times of macro-benchmarks controlled and not controlled by DiscNice. The threshold was set to infinity.

Table 8 lists the results. The overhead is less than 4.0% in `tar` and `grep`. The one reason for the overhead is the use of Linux's unsophisticated utilities of interprocess communication. With the improvement of these utilities, we could lower the overhead. The overhead of `make` is higher than those of other benchmarks. Since `make` uses CPU time more frequently than disk bandwidth, DiscNice has a greater impact on performance. We can see that the overhead incurred by DiscNice is less than 12.0%.

Note that these overheads include the side

Table 9 Comparison of execution times of `sequential`.

Benchmark	Execution time [sec]	Increasing rate
w/o `rsync`	4.18	—
w/- controlled `rsync` (1 MB)	4.81	106%
w/- controlled `rsync` (3 MB)	5.91	143%
w/- controlled `rsync` (5 MB)	8.27	201%
w/- low priority `rsync`	17.26	413%
w/- `rsync` [sec]	17.49	418%

Table 10 Comparison of execution times of `tar`.

Benchmark	Execution time [sec]	Increasing rate
w/o `rsync`	4.47	—
w/- controlled `rsync` (1 MB)	5.02	112%
w/- controlled `rsync` (3 MB)	6.73	151%
w/- controlled `rsync` (5 MB)	9.15	205%
w/- low priority `rsync`	12.82	287%
w/- `rsync` [sec]	13.10	293%

effect of the cache detector. As described in Section 5.4, the cache detector affects the performance of the target application because the probes by the cache detector disturb the read-ahead mechanism of the underlying Linux. In our benchmarks, `tar` is the worst case because it reads a 200 MB sequentially. But the overhead of `tar` is less than 4.0%.

6.5 Unobtrusiveness of Background Applications

To demonstrate that DiscNice can make background applications unobtrusive, we conducted an experiment similar to that described in Section 2.1. We measured the execution time for `sequential` and `tar` with the background application `rsync` controlled by DiscNice. The threshold was set to 1 MB/s, 3 MB/s, and 5 MB/s.

Table 9 lists the execution times for `sequential`. We can see that the execution time for `sequential` with uncontrolled `rsync` is about three times longer than that without `rsync`. By controlling `rsync`, the execution time improves dramatically. When the threshold is set to 1 MB, the increase in execution time is about 6%. When the threshold is set to 5 MB, the increase is about 101%, which is much less than the 318% in the uncontrolled case.

Table 10 lists the execution times for `tar`. Similarly to the case of `sequential`, the execution time is improved by controlling `rsync`. When the threshold is set to 1 MB, the increase in the execution time is about 12%. When the threshold is set to 5 MB, the increase is about 105%. These results suggest that DiscNice can prevent a background application from obtruding on the user's active jobs.

7. Related Work

Some applications, such as SETI@home [13] and Folding@home [10], attempt to avoid resource contention by only running themselves when the screen saver is running. This approach relies on the assumption that the system is idle if and only if the screen saver is running. This assumption is not reasonable because the PC user's jobs may be active even if the screen saver is running. In addition, numerous PC resources are often idle even if the screen saver is *not* running.

Rate-window [18] is a scheme for restricting I/O performed by a target application. It is implemented as a loadable kernel module. The disk I/O size, in the rate-window, is calculated from the ratio of file I/O and disk I/O requests. The disk I/O size is obtained from the `proc` file system. The disk I/O size obtained here is not for each process but the total size of disk I/O incurred by all the processes. Therefore, it is difficult to determine the disk I/O size for each process. If two processes, whose disk I/O ratios are vastly different, compete with each other for disk bandwidth, the calculated disk I/O sizes are erroneous. As a result, if the actual disk I/O size of the target is larger than the calculated disk I/O size, the target process is not sufficiently throttled and the performance of the user's active job deteriorates.

MS Manners [7] is a mechanism that employs progress-based regulation [19] to prevent low-importance processes from degrading the performance of high-importance processes. MS Manners monitors the progress of low-importance processes and determines when they should politely defer to a high-importance process. In MS Manners, the user cannot specify the amount of resources that a background application can borrow from the PC user.

Entropia [6] is a desktop grid system running on Windows, which monitors and limits the application use of a variety of important resources such as the CPU, the memory, and the disk. Entropia limits the amount of disk use but does not control the disk bandwidth.

A user-level sandbox [5] enforces quantitative restrictions on the use of resources by applications. Newhouse and Pasquale [14] developed a user-level scheduler that allows the user to con-

trol the CPU time. Neither of these supports the control of the disk bandwidth.

Much work has been done in the area of process scheduling and resource control [8),15),20),22)] in the context of the OS kernel. Disk I/O could be gracefully controlled if it is implemented at the kernel-level because the kernel could know the exact size of disk I/O with a minimum overhead. Our contribution is that the disk I/O can be controlled at the user-level with a modest overhead and greater precision.

8. Conclusion

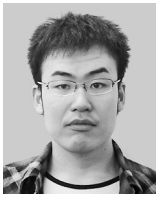
A background application such as housekeeping and PC Grid shares the resources of a user's PC to accomplish a given task. The background application often competes with and degrades the performance of the user's active jobs. In this paper, we presented DiscNice, a user-level mechanism of controlling the disk bandwidth; it prevents background applications from competing for the disk bandwidth. DiscNice predicts disk I/O behavior in the kernel from the file I/O size and from knowledge of the underlying OS. By doing this, DiscNice can regulate disk I/O at the user-level. The prototype implementation of DiscNice runs on Linux 2.4.27, and the experimental results suggest that the DiscNice can control disk I/O with a modest overhead and a great precision.

References

1) The KaZaA Media Desktop. http://www.kazaa.com
2) Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H.: Information and Control in Gray-Box Systems, *Proc. ACM Symposium on Operating Systems Principles*, pp.43–56 (Oct. 2001).
3) Butt, A.R., Gniady, C. and Hu, Y.C.: The performance impact of kernel prefetching on buffer cache replacement algorithms, *Proc. 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp.157–168 (June 2005).
4) Cantrill, B., Shapiro, M.W. and Leventhal, A.H.: Dynamic Instrumentation of Production Systems, *Proc. USENIX 2004 Annual Technical Conference* (*USENIX '04*), pp.15–28 (June 2004).
5) Chang, F., Itzkovitz, A. and Karamcheti, V.: User-level Resource-constrained Sandboxing, *Proc. 4th USENIX Windows System Symposium* (*WSS 2000*), pp.25–36 (Aug. 2000).
6) Chien, A., Elbert, B.C.S. and Bhatia, K.: Entropia: Architecture and performance of an enterprise desktop grid system, *Journal of Parallel and Distributed Computing*, Vol.63, pp.597–610, 2003.
7) Douceur, J.R. and Bolosky, W.J.: Progress-based regulation of low-importance processes, *Proc. 17th ACM Symposium on Operating Systems Principles* (*SOSP '99*), pp.247–260 (Dec. 1999).
8) Eggert, L. and Touch, J.D.: Idletime Scheduling with Preemption Intervals, *Proc. ACM Symposium on Operating Systems Principles*, pp.249–262 (Oct. 2005).
9) Figueiredo, R.J., Dinda, P.A. and Fortes, J.A.B.: A Case For Grid Computing On Virtual Machines, *Proc. 23rd IEEE International Conference on Distributed Computing SYSTEMS* (*ICDCS 2003*), pp.550–559 (May 2003).
10) Folding. Folding@home distributed computing. http://folding.stanford.edu/
11) Frey, J., Tannenbaum, T., Livny, M., Foster, I. and Tuecke, S.: Condor-G: A Computation Management Agent for Multi-Institutional Grids, *Proc. 10th IEEE International Symposium on High Performance Distributed Computing* (*HPDC '01*), pp.55–63 (Aug. 2001).
12) Gupta, A., Lin, B. and Dinda, P.A.: Measuring and Understanding User Comfort With Resource Borrowing, *Proc. 13th IEEE International Symposium on High Performance Distributed Computing* (*HPDC '04*), pp.214–224 (2004).
13) Korpela, E., Werthimer, D., Anderson, D., Cobb, J. and Lebofsky, M.: SETI@HOME-Massively Distributed Computing for SETI, *IEEE Computing in Science and Engineering*, Vol.3, No.1, pp.78–83 (2001).
14) Newhouse, T. and Pasquale, J.: ALPS: An Application-Level Proportional-Share Scheduler, *Proc. IEEE International Symposium on High Performance Distributed Computing*, pp.279–290 (June 2006).
15) Nieh, J., Vaill, C. and Zhong, H.: Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler, *Proc. 2001 USENIX Annual Technical Conference*, pp.245–259 (June 2001).
16) Ripeanu, M., Foster, I. and Iamnitchi, A.: Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design, *IEEE Internet Computing Journal*, Vol.6, No.1, pp.50–57 (Feb. 2002).
17) Ryu, K.D., Hollingsworth, J.K. and Keleher, P.J.: Exploiting Fine Grained Idle Periods in Networks of Workstations, *IEEE Transactions on Parallel and Distributed*, Vol.11, pp.683–698 (2000).

18) Ryu, K.D., Hollingsworth, J.K. and Keleher, P.J.: Efficient Network and I/O Throttling for Fine-Grain Cycle Stealing, *Proc. ACM/IEEE Conference on Supercomputing* (*SC '01*) (*CDROM*), pp.3–3 (Nov. 2001).
19) Steere, D.C., Goel, A., Gruenberg, J., McNamee, D., Pu, C. and Walpole, J.: A Feedback-driven Proportion Allocator for Real-Rate Scheduling, *Proc. 3rd Symposium on Operating System Design and Implementation* (*OSDI '99*), pp.145–158 (Feb. 1999).
20) Sullivan, D.G. and Saltzer, M.I.: Isolation with Flexibility: A Resource Management Framework for Central Servers, *Proc. 2000 USENIX Annual Technical Conference*, pp.337–350 (June 2000).
21) The Apache Software Foundation, Apache HTTP server, 1995. http://www.apache.org/
22) Waldpurger, C.A. and Weihl, W.E.: Stride Scheduling: Deterministic Proportional-Share Resource Management, Technical report, MIT/LCS/TM-528, Massachusetts Institute of Technology (1995).

(Received May 7, 2007)
(Accepted September 25, 2007)



Hiroshi Yamada was born in 1981. He received his B.E. and M.E. degrees from University of Electro-Communications in 2004 and 2006, respectively. He is currently a Ph.D. candidate in School of Science for Open and Environmental Systems at Keio University, since 2006. His research interests include virtual machine technology, operating systems, and system software. He is a member of IPSJ, IEEE/CS and ACM.

Kenji Kono received the BSc degree in 1993, MSc degree in 1995, and Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE/CS, ACM and USENIX.