Caching for Bursts (C-Burst): Let Hard Disks Sleep Well and Work Energetically

Feng Chen and Xiaodong Zhang
Dept. of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210, USA
{fchen, zhang}@cse.ohio-state.edu

ABSTRACT

High energy consumption has become a critical challenge in all kinds of computer systems. Hardware-supported Dynamic Power Management (DPM) provides a mechanism to save disk energy by transitioning an idle disk to a low-power mode. However, the achievable disk energy saving is mainly dependent on the pattern of I/O requests received at the disk. In particular, for a given number of requests, a bursty disk access pattern serves as a foundation for energy optimization. Aggressive prefetching has been used to increase disk access burstiness and extend disk idle intervals, while caching, a critical component in buffer cache management, has not been paid a specific attention. In the absence of cooperation from caching, the attempt to create bursty disk accesses would often be disturbed due to improper replacement decision made by energy-unaware caching policies. In this paper, we present the design of a set of comprehensive energy-aware caching schemes, called *C-Burst*, and its implementation in Linux kernel 2.6.21. Our caching schemes leverage the 'filtering' effect of buffer cache to effectively reshape the disk access stream to a bursty pattern for energy saving. The experiments under various scenarios show that C-Burst schemes can achieve up to 35% disk energy saving with minimal performance loss.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Main Memory

General Terms

Design, Experimentation, Performance

Keywords

Hard disk, buffer caches, energy saving, power management

1. INTRODUCTION

Efficient power management has become a required consideration in advanced computing system design. According to a recent report [8], the overall power consumption by computer servers in data centers has doubled between 2000 and 2005, and this number is still projected to increase in the near future. Similarly, energy is also regarded as a critical yet limited resource for mobile computing devices. The explosively increasing need for energy in all kinds of systems demands that optimization of energy consumption be a top priority in system design and implementation. In this paper we present a novel system enhancement in buffer cache management to significantly improve energy efficiency in storage systems.

In a computer system, the hard disk is one of the major contributors to the overall energy consumption, particularly for data-intensive applications. For example, disk drives contribute as high as 86% of the total energy consumption in a typical EMC Symmetrix 3000 storage system [7]. In order to save disk energy, most practical systems adopt a time-out based strategy, called *Dynamic Power Management*: when a disk is idle for a specific period of time (time-out threshold), it is spun down to save energy. Upon arrival of a request, the disk is spun up to service the request. To justify a substantial energy and performance overhead for spinning up/down disk, the hard disk must stay in the standby mode for a sufficiently long period of time, called *break-even time*. Therefore, the power consumption in disks can be most effectively optimized only if requests to disks are clustered in bursts with long idle intervals in between. Increasing *burstiness* of disk accesses is the key to improving efficiency of disk energy consumption.

Prefetching has been recognized as an effective mechanism for increasing disk access burstiness [15]. By pre-loading to-be-used data into memory, the future disk accesses can be directly 'condensed' into a sequence of I/O bursts. However, caching as a fundamental system component in buffer cache management has not been paid a specific attention. Without coordination and a specific effort from an energy-aware caching design, buffer cache management would have the following limits and cause disk energy saving to be sub-optimal.

1. **Caching operations can significantly affect and disturb periodic bursty patterns in disks.** By improperly selecting a victim block for eviction, a caching policy without energy awareness can easily foil the effort made by prefetching for organizing a periodic bursty access pattern. On the contrary, an energy-aware caching policy can coordinate well with prefetching to maximize the idle intervals and create a bursty access pattern.

2. **Aggressive prefetching shrinks available caching space, thus demands highly effective caching decisions.** In order to maintain sufficient coverage on future data requests, all likely-to-be-accessed blocks need to be prefetched into memory, which raises high memory contention. As a result, the cache replacement mechanism is often activated to free memory. Properly selecting a block for eviction becomes increasingly critical to system-wide energy saving.

3. **Energy-aware caching policy can effectively complement prefetching.** Prefetching can be effective only when its prediction on future accesses is correct. In real systems it is difficult to achieve required accurate prediction, especially when handling workloads with complicated access patterns. When prefetching works unsatisfactorily, energy-efficient caching policies can well complement prefetching besides playing a coordinating role only.

Most existing caching algorithms, such as the well-known LRU algorithm and recently proposed CLOCK-Pro [13], are designed for improving performance only. They cannot be directly employed for the purpose of energy saving for two reasons. First, they are originally designed for reducing the number of disk accesses with no consideration of increasing disk access burstiness. Second, these algorithms usually do not pay a specific attention to the physical time of disk accesses, which

ISLPED'08, August 11–13, 2008, Bangalore, Indian..
Copyright 2008 ACM 978-1-60558-109-5/08/08 ...$5.00.

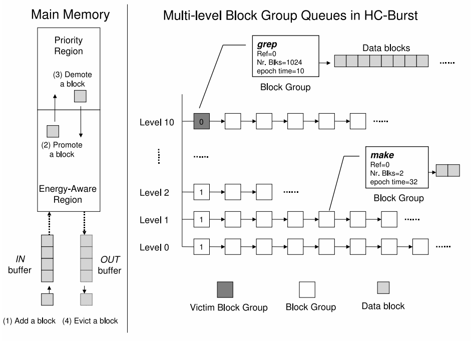


Figure 1: The HC-Burst Caching Scheme.

is essential for estimating energy consumption. In this paper, we present a set of comprehensive energy-aware caching schemes and its implementations in Linux kernel, called *C-Burst* (Caching for Bursts). By leveraging the 'filtering' effect of buffer cache, we can effectively reshape the disk access stream to an expected bursty pattern and substantial energy saving can be achieved.

As an executive summary, we have made the following contributions: (1) We present two comprehensive energy-aware caching policies, the *History-based C-Burst (HC-Burst)* and the *Prediction-based C-Burst (PC-Burst)*, and their implementations in Linux. (2) Our schemes do not rely on complicated disk power models and require no disk specification data. Also, we do not assume any specific disk hardware, such as the previsioned multi-speed disks [9]. (3) We provide flexible performance guarantees to maintain reasonable performance degradation under a tolerable performance loss rate. (4) Unlike many previous simulation based research work, we have implemented our algorithms in recent a Linux kernel 2.6.21.5 and the experiment results show that C-Burst schemes can achieve up to 35% disk energy saving with minimal performance loss.

2. DESIGN OF C-BURST SCHEMES

In this section we will first introduce the main architecture of our buffer cache management, then we present the design of HC-Burst and PC-Burst schemes in details.

2.1 Priority Region & Energy-Aware Region

Buffer cache management can affect system performance significantly. In order to avoid raising overwhelming memory misses, the 'hot' blocks should be safely held in memory. Thus, we segment the buffer cache space into two areas, *Priority Region* and *Energy-Aware Region*. The blocks with strong locality (hot blocks) are managed in the priority region using a traditional locality-based replacement policy, such as the 2Q-like replacement algorithm in Linux, to minimize the number of memory misses. The blocks with weak locality (cold blocks) are managed in the energy-aware region using our energy-aware replacement schemes. The partition size of priority region is initially set half of the total buffer cache space and automatically tuned on the fly (see Section 2.4).

As shown in Figure 1, when a new block is added into buffer cache, it is first added into the energy-aware region, because it has never been accessed before and shows weak locality. An *IN* buffer is used to collect blocks and insert them into the energy-aware region in cluster. Once a block is reaccessed, it is promoted into the priority region as a hot block. Accordingly, a block in the priority region is demoted to the energy-aware region. Whenever free memory is needed, a victim block is always selected from the energy-aware region using our energy-aware replacement policies, HC-Burst or PC-Burst scheme.

2.2 History-based C-Burst Scheme

2.2.1 Main Idea

Most computer systems can simultaneously run multiple tasks, which may exhibit significantly different access patterns. Unfortunately, the existing buffer cache management does not consider such fundamental divergence and handles data accessed by various tasks in the same way. In order to increase disk access burstiness and save disk energy, the access pattern of each task should be characterized individually and the accessed data blocks should be managed accordingly. For example, in a system concurrently running *grep*, a text search tool, and *make*, a compiling tool, purposely holding the dataset of *make* in memory and aggressively evicting that of *grep* is a sensible choice, since *grep* can load data from disk in a short burst, which incurs a minimal energy cost.

2.2.2 Tracking Tasks

A task's access pattern can be recognized based on its access history, however, many tasks' lifetimes are too short to be tracked, say a few seconds or even less. To address this problem, we associate an *I/O Context (IOC)* to each task to maintain its access history across many runs. IOCs are managed in a hash table and each IOC is identified by a unique *ID* number, which is a 32-bit hash value of the absolute pathname of a user-level application's executable or the task name of a kernel thread.

2.2.3 Identifying Burstiness

The access pattern of a task may change over time. Thus we break down the disk operation time to epochs, say T seconds for each. Selecting a proper epoch length, T, is non-trivial. A too short or a too long epoch are both undesirable. We suggest to adopt half of the disk spin-down time-out threshold as the epoch length for two reasons. First, by comparing two epoch times, we can easily *infer* whether a disk spin-down, the most critical event for disk energy saving, would occur between them or not. Second, we can *ignore* the distribution of individual I/O requests in each epoch, because disk energy consumption would not change as long as no disk power transition happens between requests.

In each epoch all the data accesses generated by a task are called an *I/O burst* in aggregate. The blocks requested in one I/O burst are managed as a unit, called *Block Group (BG)*. Each block group is identified by the task's IOC ID and the epoch time. The blocks in a block group are managed in the LRU order. In this way, a task's access pattern can be described using a sequence of block groups, and the disk access *burstiness* during each epoch can be described using the number of blocks in the corresponding block group.

2.2.4 HC-Burst Replacement Policy

When free memory is needed, a victim block should be identified for replacement. Two kinds of blocks are of special interests, the blocks being accessed in a bursty pattern and blocks that are unlikely to be reaccessed. Such blocks can be found in the largest block group. This is because the more burstily a task accesses data during an epoch, the larger the block group would be. Also, the less frequently these blocks are used, the less number of blocks in the block group would be promoted to the priority region. Therefore, we need to identify the largest block group as a victim block group first and return the LRU block as a victim block.

In order to efficiently identify the victim block group, we maintain a structure of multi-level queues to manage the block groups as shown in Figure 1. Each queue links a number of block groups in the order of their epoch times. Block groups sharing the same epoch time but owned by different tasks are placed together in their insertion order. Each block group can stay in only one queue, and the queue level is determined by the size of the block group. Specifically, for a block group with N blocks, the queue level it stays at is $\lfloor log_2(N) \rfloor$. When the block group's size changes (e.g. a block is promoted to or demoted from the priority region), the block group may move upwards or downwards on the queue stack. We maintain 32 queues in total, and block groups containing equal to or more than 2^{31} blocks are placed on the top queue.

Identifying a victim block group is simple. We scan from the top queue level to the bottom. If the queue contains valid block groups, we select the block group with the oldest epoch time (the LRU block group) as a victim. All of its blocks are filled to an *OUT* buffer for replacement, once the buffer becomes half empty, victim block groups are identified to refill the buffer. The blocks in the buffer are evicted in their insertion order.

In order to avoid holding inactive block groups at low levels infinitely, we associate each block group with a reference flag, which is cleared initially. Whenever a block is accessed, the reference flag of its block group is set to indicate that this block group is being actively used. Each time when a victim block group at queue level q is evicted from memory, we scan the LRU block groups on queues from level $q-1$ to level 0. The first met block group with unset reference flag is identified as a victim; otherwise, the block group's reference flag is cleared. In this way, the blocks that have not been accessed for a while will be gradually evicted.

2.3 Prediction-based C-Burst Scheme

2.3.1 Main Idea

Caching policy can not only affect the disk access *burstiness* but also manipulate the *timing* of disk accesses. Selectively evicting or holding a block which is to be accessed at a specific time will create or remove a disk access at that time. Suppose we know some disk accesses that will deterministically occur in the future, namely *deterministic accesses*, we can purposely coordinate with these known future disk accesses to improve disk energy efficiency. For example, we can evict a block that will be accessed during a short interval between two deterministic accesses and avoid breaking a long idle interval. The challenge is how to precisely and efficiently predict the deterministic accesses and blocks' access times.

2.3.2 Predicting Deterministic Accesses

Predicting future disk accesses is challenging. Some previous research work [2, 5] adopts complicated models, such as learning tree and Markov processes, to predict disk activities. Unfortunately, directly applying such models in real systems often brings prohibitively high overhead and works unsatisfactorily to accommodate uncertain dynamics. Here we present a practical solution to identify the deterministic accesses.

```
Initialize credit = 0;
Initialize predictable = 0;
Initialize cred_inc = cred_dec = 1;

/* when an idle interval is observed */
if observed interval == predicted interval
  credit   = min(credit+cred_inc, 32);
  cred_inc = min(cred_inc*2, 8);
  cred_dec = max(cred_dec/2, 1);
else
  credit   = max(credit-cred_dec, -32);
  cred_dec = min(cred_dec*2, 8);
  cred_inc = max(cred_inc/2, 1);

predictable = (credit > 0) ? TRUE : FALSE;
```

Figure 2: Prediction of deterministic accesses

In a computer system, there exist many tasks that periodically request disk data in a well predictable pattern. For example, many system kernel threads are triggered by built-in timers and periodically access disk (e.g. *pdflush* writes dirty blocks to disk every 5 seconds). Also, many applications, especially multimedia applications, have a steady data consumption rate and their disk accesses are well predictable [6]. Such periodically incurred disk accesses can serve as deterministic accesses.

In practical environment, however, intervals between such periodic disk accesses may still change over time. It could be caused by true pattern shift or just occasional system dynamics. We need to respond quickly to real pattern change and accommodate accidental deviation simultaneously. We present a simple yet effective algorithm as shown in Figure 2. For each task we predict the future interval using the average of four recently observed intervals. Once a new interval is observed, it is compared with the predicted one. If the prediction is continuously proved to be correct, we credit the task exponentially. When significant pattern change occurs, consecutive mis-prediction should be observed and the task's credit would be quickly decreased. Meanwhile, occasional dynamics would just charge the task's credit slightly. Once the task's credits become negative, it is identified as 'unpredictable'. Using predicted interval, we can easily estimate a task's future disk accesses. The predicted disk accesses of all predictable tasks are merged to get system-wide deterministic accesses.

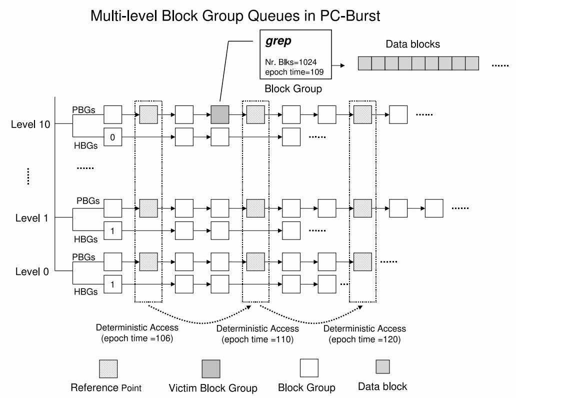


Figure 3: The PC-Burst Caching Scheme.

2.3.3 Predicting Block Re-accesses

Estimating a block's future access time is even more challenging, since we have to maintain access history of all blocks, including those non-resident blocks. We use the block table, which has been effectively used in DULO [14], to record up to four epoch times for each block with little overhead. Interested readers can refer to the paper for more details. By searching the block table with logic block number (LBN), we can quickly locate the epoch times of recent accesses to a block. If the block has been accessed in a constant interval, we can easily predict the next access. Otherwise, this block is recognized as 'unpredictable'. The reason we take such a conservative way for prediction is that, improperly evicting a block may cause a disk access at an unexpected time (e.g. in the middle of a long disk idle interval), which leads to undesired energy expense.

2.3.4 PC-Burst Replacement Policy

In PC-Burst scheme, we introduce *Predicted Block Groups (PBG)* to hold blocks that are predicted to be accessed at a future epoch time. The remaining unpredictable blocks are managed in *History Block Groups (HBG)* as in HC-Burst scheme. As time elapses, a predicted block group may expire (the predicted epoch time is passed), its associated blocks are moved to corresponding history block groups.

As shown in Figure 3, the PC-Burst scheme also maintains 32-level queues and each level has two queues, one for predicted block groups and the other for history block groups. Block groups are placed on the queues in the same way as in HC-Burst scheme. In the queue of predicted block groups, *Reference Points (RP)* are inserted to represent deterministic accesses. To facilitate a quick scan, the reference points on the same queue are linked together.

For replacement, we start from the top queue level to the bottom level. On each level, the queue of predicted block groups is examined first. We traverse all reference points linked in the queue and attempt to locate a victim from predicted block groups. For each reference point, we check its neighbor block groups on the queue. If the epoch time of the predicted block group is exactly the same as the reference point or just one epoch before or after the reference point, it is identified as a victim. If no such block group exists, the block group locating in the shortest interval between two reference points is selected. If multiple such block groups are found, the one whose epoch time is closest to a reference point is preferred. Whenever possible, the block group to be accessed in the farthest future is identified as a victim first. If no predicted block group exists on the queue, the history block groups are checked using the same policy as in the HC-Burst scheme.

2.4 Performance Loss Control

Energy-aware caching policies may introduce more memory misses by aggressively replacing blocks that are *recently* accessed but in a *bursty* pattern. To estimate the incurred memory misses due to applying energy-aware caching, we maintain a pseudo-LRU buffer, whose size is the same as the energy-aware region, and simulate LRU replacement policy there. Once a memory miss occurs, we first check if the block is resident in the pseudo-LRU buffer. If true, the memory miss is identified as a miss caused by applying energy-aware caching, because if the LRU replacement policy was conducted, the memory miss would be avoided. Every T seconds, we estimate a total performance loss, T_{loss} seconds, by multiplying the number of such memory misses with the observed average disk latency. The performance loss rate, P_{loss}, can be estimated as T_{loss}/T.

The user can set a maximum tolerable performance loss rate, which serves as a high watermark, HW. We also set a low watermark, LW, say 1%, and use MW to denote $\frac{HW+LW}{2}$. The energy-aware region size is adaptively tuned as follows. Suppose current energy-aware region size is S_{cur}, then: if P_{loss} is smaller than MW, the target energy-aware region size S_{tgt} is $(1 + min(\frac{(MW-P_{loss})\times 2}{HW-LW}, 1)) \times S_{cur}$; if P_{loss} is larger than MW, S_{tgt} is $(1 - min(\frac{(P_{loss}-MW)}{HW-LW}, 0.5)) \times S_{cur}$. The new energy-aware region size S_{next} is $S_{cur} \times p + S_{tgt} \times (1 - p)$, where p is set 0.5 to smooth the change of region size.

3. PERFORMANCE EVALUATION

To evaluate the HC-Burst and PC-Burst schemes in a practical operating system, we implemented a prototype in a Linux 2.6.21.5 kernel with about 1000 lines of code in 15 existing files and another 4500 lines of code in new files to implement the main algorithms.

3.1 Implementation Issues

In order to predict the deterministic disk accesses, we need to track the disk I/O operations caused by each task. In Linux, requests for disk data are first issued to the generic block layer, then forwarded to the I/O scheduler, which later issues requests to the disk. Directly associating each operation observed at disk with one specific task is cumbersome, since I/O scheduler may merge, sort, and reschedule I/O requests issued from tasks. Instead, we monitor the requests issued by each task on the generic block layer. Though there may exist a small delay between when the request is received at generic block layer and when the disk actually starts transferring requested blocks, the interval is nearly negligible compared to the epoch length (usually a few seconds) and this works well in practice.

3.2 Experiment Methodology

The experiments were conducted on a machine with a 3.0GHz Intel Pentium 4 processor, 1024MB memory, and a Western Digital WD1600JB 160GB 7200rpm hard drive. The OS is Redhat Linux WS4 with the Linux 2.6.21.5 kernel using the Ext3 file system.

To evaluate the disk energy consumption, we adopt a methodology similar to [11]. The disk activities are collected in an experiment machine and sent via *netconsole* to another monitoring machine through a Gigabit network interface. Using the collected trace of disk accesses, we calculate disk energy consumption based on the disk power models off line. Two disk models are emulated in our experiments, a HITACHI DK23DA laptop disk [10, 15], which features 30GB capacity, 2MB cache, 4200 RPM, and 35MB/sec bandwidth, and an IBM Ultrastar 36Z15 SCSI disk [12, 16], which has 18.4 GB capacity, 4MB cache, 15000 RPM, and 53MB/sec bandwidth. Table 1 shows the energy consumption parameters.

With direct support from the Linux kernel, Linux laptop-mode [1] is designed for optimizing disk energy consumption by employing many techniques, such as delayed write-back and aggressive prefetching. As the de facto disk energy-saving mechanism used in practice, Linux laptop-mode (denoted as *LPM*) is used as the baseline policy to compare with our *HC-Burst* and *PC-Burst* schemes. Since our schemes are designed for optimizing single disk energy consumption and largely orthogonal to the multi-disk oriented caching policies, such as PA-LRU [16] and PB-LRU [17], we did not implement and compare with them. Also, in experiments we do not assume using the previsioned multi-speed disks [9] as these studies.

Power mode	HITACHI-DK23DA	IBM Ultrastar
Active Power	2.00 W	13.5 W
Idle Power	1.60 W	10.2 W
Standby Power	0.15 W	2.5 W
Spin up	1.6 sec/5.00 J	10.9 sec/135 J
Spin down	2.30 sec/2.94 J	1.5 sec/13 J

Table 1: The disk energy consumption data.

In the experiments, the write-back interval is set 120 seconds, the highest ratio of dirty blocks is set 60%, the lowest ratio is set 1%. The readahead is set 6144 sectors. The disk spin-down timeout is 10 seconds, and accordingly the epoch length is 5 seconds. The maximum tolerable performance loss rate is set 30%, and the interval of tuning region size is set 30 seconds.

3.3 Workloads

Name	Description	MB/ep.	Req./ep.
make	Linux kernel compiler	1.98	119.7
vim	text editor	0.006	0.395
mpg123	mp3 player	0.15	3.69
transcode	video converter	3.2-6.5	10.9-19.1
TPC-H	database query 17	7.3	476.7
grep*	textual search tool	102.2	10186.6
scp*	remote copy tool	51.5-53.8	135-139
CVS*	version control tool	19.9	1705.7

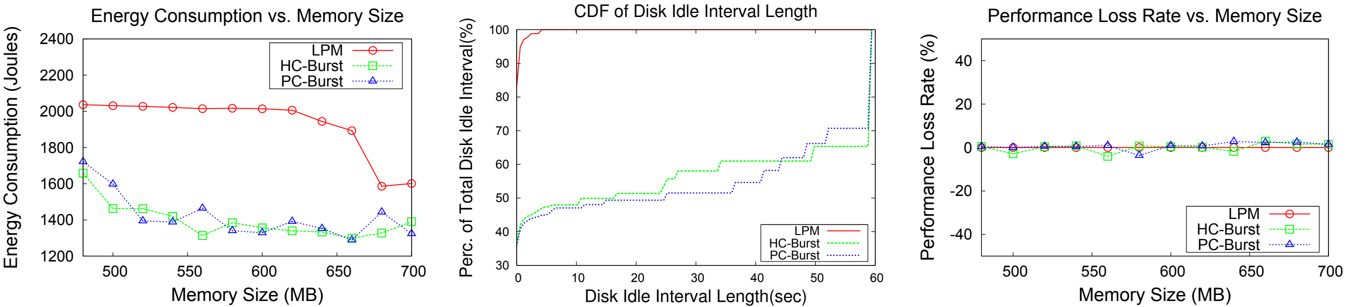
Table 2: The application description. Applications with bursty pattern are denoted with (*). The table shows the amount of accessed data and the number of requests per epoch for each application. Data are collected in the default system setting.

As listed in Table 2, five applications with non-bursty access pattern and another three applications with bursty access pattern are used in our experiments. We run these applications in combination to synthesize three representative scenarios, *programming*, *multimedia processing*, and *multi-role server* for our case study. Since energy consumption is time sensitive, we carefully concatenate applications in sequence and use shell scripts to accurately control the timing of each workload for repeatable experiments. The details about each workload are presented in later sections.

3.4 Case Study I: Programming

The first case emulates a typical software development scenario, and three applications, *make*, *grep*, and *vim*, are involved. In this workload, the user is working on Linux kernel by building the kernel image three times in sequence. When compiling kernels, the user is editing another set of Linux source code simultaneously using *vim*. Every three minutes the user searches the source code for keywords that are randomly selected from file */boot/System.map*.

As shown in Figure 4(a), HC-Burst and PC-Burst consume much less energy than LPM, even with limited memory space. For example, with 480MB memory, LPM consumes 2036.6J energy, while HC-Burst and PC-Burst consume 18.5% less (1658.6J) and 15.4% less (1721.8J) energy respectively. As memory size increases, the disk energy consumption for LPM does not show corresponding improvement until available memory size increases over 660MB. This is because LPM uses locality-based replacement policy without considering task access pattern. As long as available memory space cannot hold the working set of both *make* and *grep*, searching a large amount of data using *grep* will evict partial working-set of *make* from memory. This incurs constant disk accesses later. In contrast, C-Burst schemes protect working set of



(a) Energy consumptions (b) CDF of Disk Idle Interval Length (600MB) (c) Performance Loss Rate.

Figure 4: *Programming*: Energy consumption, CDF of disk idle period length, and performance loss rate

make in memory and only incur bursty accesses for *grep*. As a result, C-Burst schemes are more likely to benefit from increased memory size. For example, HC-Burst and PC-Burst consume around 18.2% less (1356.6J) and 22.8% less (1329.1J) energy with 600MB memory than with 480MB memory. Note that if memory size is too small, say less than the working set size of *make*, our schemes would have limited space to effectively protect *make* and work no better than LPM. However, it is clear that our solution can more effectively leverage limited available memory space to achieve higher energy efficiency.

To further explain how C-Burst schemes increase disk access burstiness, we plot the Cumulative Distribution Function (CDF) curves of the disk idle interval length, as shown in Figure 4(b). For LPM, nearly 98.8% disk idle intervals are shorter than 3 seconds. In contrast, HC-Burst and PC-Burst significantly extend disk idle intervals (50.2% and 50.7% of disk idle intervals are longer than 16 seconds, the *break-even time*).

As shown in Figure 4(c), little performance loss is observed with HC-Burst and PC-Burst schemes. This is because, selectively holding the dataset of *make* and biasing *grep* not only tailor disk accesses to a bursty pattern but also reduce memory misses. For instance, with 600MB memory, LPM issues 186,092 disk requests, while HC-Burst has only 148,770 disk requests. The reduction of memory misses effectively compensates the disk power mode transition overhead.

3.5 Case Study II: Multimedia Processing

Multimedia applications are very popular in mobile computing environment, and the second case emulates a multimedia processing scenario. In this case, the user uses *transcode* to convert a 160MB mpeg movie file to divx format files with three different frame size setting (480x360, 320x240, 240x180). After each conversion, the file is uploaded to a remote media server via *scp*. While waiting for transcode to finish, the user is using *mpg123* to play a 103MB mp3 file till movie conversion and uploading end. In this case Linux readahead is set 12288 sectors to make aggressive prefetching.

Overshooting prefetching is harmful. When memory is scarce, prefetched blocks could be evicted even before they are actually used, which is called *prefetch thrashing*. In Linux once such a situation is observed, prefetching is slowed down to operate conservatively, which unfortunately results in continuous disk accesses later. As shown in Figure 5(a), with only 600MB memory, all three schemes suffer from prefetch thrashing. As memory size increases, effectiveness of prefetching improves, and C-Burst schemes benefit the most. For example, with 700MB memory, LPM consumes 1164J energy, while HC-Burst and PC-Burst consume only 1075.1J and 934J energy, respectively. This is because C-Burst schemes selectively bias *scp*, which shows bursty pattern, and leave more room for prefetching blocks of *mpg123*. This effectively avoids prefetch thrashing. We further vary the prefetch depth, as shown in in Figure 6. We can see that, as prefetch depth increases from 2048 to 4096 sectors, energy consumption for LPM even increases by 8.2%. In contrast, by managing caching blocks more efficiently, HC-Burst and PC-Burst schemes can accommodate *deeper* prefetching and achieve additional energy saving. This indicates that, our schemes are especially useful to exploit potential of aggressive prefetching in a system with limited memory. As available memory size increases, prefetch thrashing risk is highly reduced and all three schemes achieve comparable energy consumption with 800MB memory.

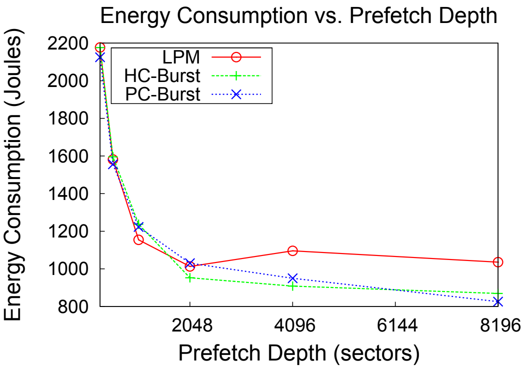


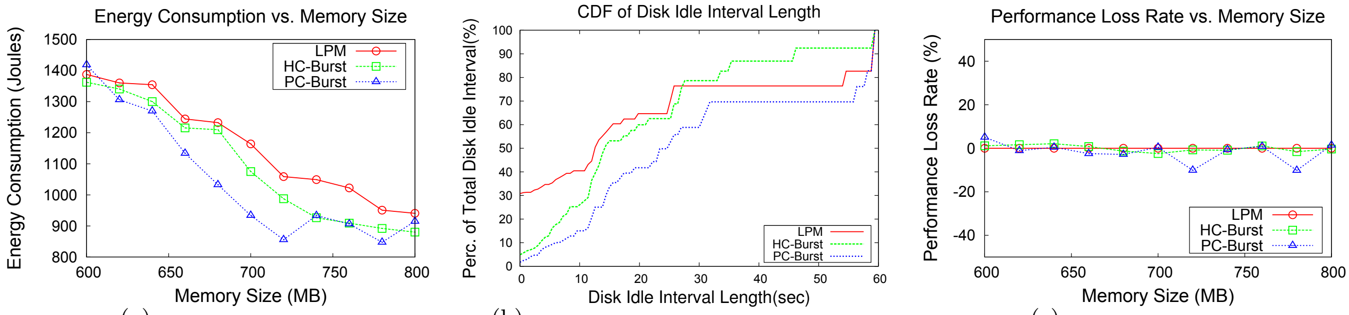
Figure 6: Disk energy consumption vs. prefetch aggressiveness (700MB memory)

In this case, we also can see that PC-Burst achieves better energy saving than HC-Burst, because this workload is dominated by sequential accesses with well predictable pattern. For example, with 700MB memory, PC-Burst has 64.6% disk idle intervals longer than 16 seconds and HC-Burst has only 46.9% idle periods longer than that, which indicates that PC-Burst can exploit more energy saving opportunities. As a result, PC-Burst consumes 13.1% less energy than HC-Burst.

3.6 Case Study III: Multi-role Server

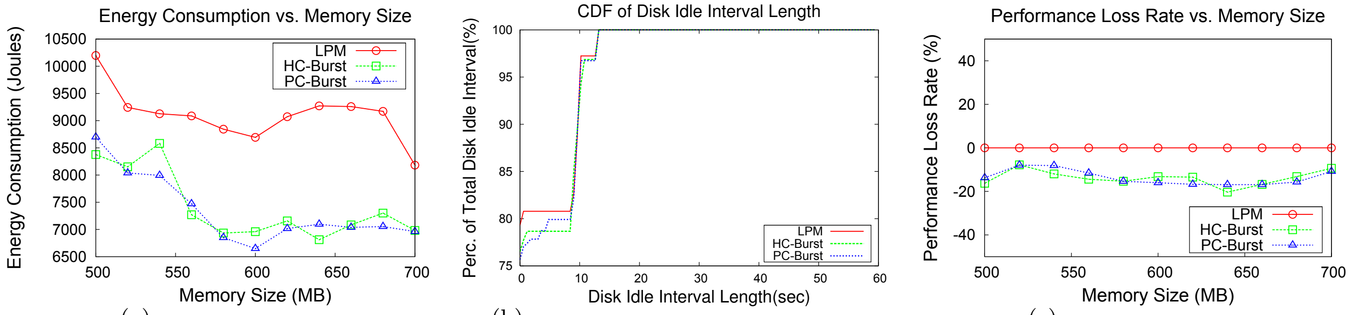
In this case, we examine the performance of HC-Burst and PC-Burst in a server environment. The disk emulated is an IBM Ultrastar 36Z15 SCSI hard disk. This server plays two major roles: It runs a PostgreSQL 7.3.18 database server to serve business queries. We choose the scale factor 1 to generate the database and run TPC-H query 17 as a representative random workload against the database server for four times. We did not run the entire TPC-H benchmark suite because of the prohibitively long run time. This server also works as a CVS server, a version control utility widely used in software development environment. After each database query, a remote user connected in a Gigabit Ethernet LAN checks out Linux kernel 2.6.21 source code of 292MB.

As shown in Figure 7(a), HC-Burst and PC-Burst consume much less energy than LPM. For example, with 600MB memory, HC-Burst and PC-Burst consume only 6960.1J and 6648.7J respectively, which are 19.9% and 23.5% less than that for LPM (8693.4J). Interestingly, Figure 7(b) does not show extension of disk idle intervals as expected. This is because, TPC-H queries characterize very random access pattern, which leads to substantial disk I/O latency. C-Burst schemes evict dataset of CVS aggressively and protects the working-set of TPC-H dataset in memory. This resulted in much less I/O latency and significantly shortened total execution time. In particular, the execution time for LPM is 553.7 seconds in total, and only 465 seconds for PC-Burst, including disk power mode transition overhead. This also explains why PC-Burst and HC-Burst actually improved performance by around 15%, as shown in Figure 7(c).



(a) Energy consumptions (b) CDF of Disk Idle Interval Length (700MB) (c) Performance Loss Rate

Figure 5: ***Multimedia*****: Energy consumption, CDF of disk idle period length, and performance loss rate**



(a) Energy consumptions (b) CDF of Disk Idle Interval Length (600MB) (c) Performance Loss Rate.

Figure 7: ***Multi-role Server*****: Energy consumption, CDF of disk idle period length, and performance loss rate**

4. RELATED WORK

Some previous studies have been conducted to create bursty access pattern and extend disk idle intervals in buffer cache management. Papathanasiou et al. [15] proposed a scheme to conduct aggressive prefetching for increasing disk access burstiness. However, they still adopt the traditional LRU-based replacement algorithm to manage caching space. As we show previously, C-Burst schemes can effectively complement and even improve the effectiveness of such prefetching based schemes by managing cached blocks more efficiently and reducing the risk of prefetch thrashing.

Zhu et al. proposed two caching schemes, PA-LRU [16] and PB-LRU [17], to manage buffer cache for improving energy efficiency in multi-disk systems. Their basic idea is to offer blocks from idle disks a high priority to stay in memory while biasing blocks from busy disks, such that the idle disks can be put in low power mode longer. Cai and Lu [3] further considered the energy consumption of memory by tuning partition sizes and disk rotational speed. Our C-Burst schemes differ from these solutions in two aspects. First, these multi-disk based studies attempt to create unbalanced loads between disks and save disk energy in a rather coarse granularity. Creating bursty access pattern in each individual disk is out of consideration. In this paper we show that there is plenty room for saving energy of each individual disk by distinguishing the divergent access patterns of tasks. Second, these schemes are mainly based on previsioned multi-speed disks [9], which are still unavailable in mainstream market. Our schemes do not rely on such multi-speed disks and can benefit existing hard disks. In general, our C-Burst schemes are largely orthogonal to these studies and can be applied to multi-disk platform with no significant change. Further leveraging special hardware features [4] and supporting multi-disk is under research as our future work.

5. CONCLUSION

In this paper we present two comprehensive caching policies, called *HC-Burst* and *PC-Burst*, for disk energy saving. Our schemes can effectively increase disk access burstiness and extend disk idle intervals by selectively replacing blocks based on tasks' access patterns and predicted access times. Our implementations in Linux kernel 2.6.21 shows that C-Burst schemes can achieve up to 35% energy saving in various scenarios with minimal performance loss.

6. ACKNOWLEDGMENT

We thank anonymous referees for their constructive comments. This work has been partially supported by the National Science Foundation under grants of CCF-0620152 and CCF-072380.

7. REFERENCES

[1] http://lxr.linux.no/source/Documentation/laptop-mode.txt.

[2] L. Benini, A. Bogliolo, G. A. Paleologo, and G. D. Micheli. Policy optimization for dynamic power management. In *IEEE TCADICS*, June 1999.

[3] L. Cai and Y. Lu. Power reduction of multiple disks using dynamic cache resizing and speed control. In *Proc. of ISLPED'06*, 2006.

[4] F. Chen, S. Jiang, and X. Zhang. SmartSaver: Turning flash drive into a disk energy saver for mobile computers. In *Proc. of ISLPED'06*, 2006.

[5] E.-Y. Chung, L. Benini, and G. D. Micheli. Dynamic Power Management Using Adaptive Learning Tree. In *Proc. of ICCAD'99*, Nov. 1999.

[6] G. de Nijs, W. Almesberge, and B. v. d. Brink. Active Block I/O Scheduling System (ABISS). In *Proc. of Ottawa Linux Symposium*, Ottawa, Canada., 2005.

[7] EMC. Symmetrix 3000 and 5000 enterprise storage systems product description guide. http://www.emc.com, 1999.

[8] K. Greene. Data centers' growing power demands, Feb. 2007.

[9] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic speed control for power management in server class disks. In *Proc. of ISCA'03*, 2003.

[10] HITACHI. http://www.hitachigst.com/tech/techlib.nsf/products/DK23DA_Series.

[11] H. Huang, W. Hung, and K. G. Shin. FS2: Dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proc. of SOSP'05*, Oct. 2005.

[12] IBM. http://www.hitachigst.com/tech/techlib.nsf/products/Ultrastar_36Z15.

[13] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An effective improvement of the clock replacement. In *Proc. of USENIX'05*, April 2005.

[14] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *Proc. of FAST'05*, 2005.

[15] A. E. Papathanasiou and M. L. Scott. Energy efficient prefetching and caching. In *Proc. of USENIX'04*, 2004.

[16] Q. Zhu, F. M. David, C. F. Devaraj, Z. Li, Y. Zhou, and P. Cao. Reducing energy consumption of disk storage using power-aware cache management. In *Proc. of HPCA'04*, 2004.

[17] Q. Zhu, A. Shankar, and Y. Zhou. PB-LRU: A self-tuning power aware storage cache replacement algorithm for conserving disk energy. In *Proc. of ICS'04*, 2004.