PABC: Power-Aware Buffer Cache Management for Low Power Consumption

Min Lee, Euiseong Seo, Joonwon Lee, and Jinsoo Kim, *Member*, *IEEE*

Abstract—Power consumed by memory systems becomes a serious issue as the size of the memory installed increases. With various low power modes that can be applied to each memory unit, the operating system can reduce the number of active memory units by collocating active pages onto a few memory units. This paper presents a memory management scheme based on this observation, which differs from other approaches in that all of the memory space is considered, while previous methods deal only with pages mapped to user address spaces. The buffer cache usually takes more than half of the total memory and the pages access patterns are different from those in user address spaces. Based on an analysis of buffer cache behavior and its interaction with the user space, our scheme achieves up to 63 percent more power reduction. Migrating a page to a different memory unit increases memory latencies, but it is shown to reduce the power consumed by an additional 4.4 percent.

Index Terms—Buffer cache, energy management, PAVM, SDRAM.

1 Introduction

The memory system is one of the most power-hungry components of most computing equipment. Aside from display devices, the memory system is second only to the processor in terms of power consumption in desktop and server environments. For example, approximately 40 percent of the total energy is consumed by the memory system in a mid-range IBM eServer machine [10]. Moreover, in hand-held devices such as PDAs and laptop computers, the memory system is one of the major power consumers. Memory systems will remain major energy consumers across many computing environments since they consume power continuously, in contrast to other components, and the amount of memory keeps increasing.

To save energy consumed by the memory system, Huang et al. proposed a scheme called power-aware virtual memory (PAVM), whereby active pages are allocated to the same memory unit to reduce the number of memory units that need to be activated [4], [5]. However, they considered only pages that are mapped to user address spaces. Although they could successfully deal with memory access in the user address space memory, access by the kernel is ignored.

A buffer cache is widely used due to the increasing speed gap between the hard disk and the main memory. As the total memory size increases, the buffer cache occupies a huge part of this. Furthermore, memory access to the buffer cache is known to equal more than half of that made by the kernel. Although reducing the energy consumed by the buffer cache is similar to PAVM [5], it differs in workload characteristics. This work can be understood as an extension of PAVM. However, the power-aware buffer cache (PABC) scheme adopted in this paper only uses the main concept of PAVM and does not exactly include the latter. In addition, we applied our implementation of PABC on Linux 2.6.10.

In this paper, we propose a new scheme, named PABC, to deal with energy issues raised by the buffer cache. It is based on a distinct observation that memory access to the buffer cache is mainly by file accesses that are related to an application that also generates memory access in its address space.

Our contributions can be summarized as follows: First, we identified the limitation of PAVM and showed that this limitation is due to the existence of buffer cache. Second, we devised an appropriate policy to deal with the buffer cache to save the energy consumed by the memory system. Third, we implemented PABC and its migration scheme in Linux 2.6.10 and evaluated its performance implications.

The rest of the paper is organized as follows: Section 2 describes some related work and explains PAVM. Section 3 details the limitation of PAVM we identified and the motivation for this study. In Section 4, the basic concepts and the design of PABC are explained and the evaluation is described in Section 5. Finally, Section 6 summarizes the paper and draws some conclusions.

2 Background

2.1 Memory Energy Model

For all synchronous DRAM (SDRAM) architectures, such as single-data-rate (SDR), double-data-rate (DDR), and Rambus (RDRAM), cells need to be refreshed periodically to maintain the information. However, in light of power consumption, it takes only a small amount of energy to refresh the cells. The other subcomponents, such as row and column decoders and sense amplifiers, consume most of the

- *M. Lee is with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280. E-mail: minlee@cc.gatech.edu.*
- *E. Seo, J. Lee, and J. Kim are with the Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1 Kuseong-dong, Yuseong-gu, Daejeon 305-701, Korea. E-mail: ses@calab.kaist.ac.kr, joon@kaist.ac.kr, jinsoo@cs.kaist.ac.kr.*

Manuscript received 14 Dec. 2005; revised 30 May 2006; accepted 10 Oct. 2006; published online 22 Feb. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0447-1205.
Digital Object Identifier no. XXX

energy. Therefore, by disabling some of these power-consuming subcomponents, SDRAM architectures provide several low-power operating modes. Although the information in the cell remains in the low-power operating mode, it should be transitioned to the high-power operating mode for reading or writing. However, a certain performance penalty, called the resynchronization cost, is incurred for these transitions between operating modes. This nonnegligible cost warrants a novel scheme to fully utilize such a power-saving facility.

Although the details depend on the specific memory technology, each SDR, DDR, and RDRAM has its own power control unit and power modes. The unit of power control in DDR RAM is called a rank and the unit in the RDRAM memory system is called a device, which provides finer granularity of control. RDRAM memory systems provide four power modes of interest—attention, standby, nap, and power-down. Memory controllers already use the standby mode to save energy at little cost owing to the small transition time between the attention mode and the standby mode. The nap mode is much more suitable for energy saving compared to the power-down mode owing to its large difference in cost and small difference in power [5]. Therefore, we can consider only two power modes in this case. Although DDR RAM memory systems provide more power modes, the trade-off between the resynchronization cost and energy saving makes it reasonable to consider only two of these modes, the standby mode and the self-refresh mode [4], [6].

Therefore, in this paper, we focus on DDR RAM and take a rank as the power control unit. We also consider only two power modes—on and off. These two modes may be understood as the attention mode and the nap mode, respectively, in the case of RDRAM, or standby mode and self-refresh mode in the case of DDR RAM.

Since the main memory can be viewed as an array of power control units and a rank is usually a chunk of memory that consists of thousands of consecutive physical pages, the memory is viewed as an array of ranks in this paper.

2.2 Related Work

Efforts to reduce the energy used by the memory system have been made in various domains. Using a hardware approach, Fan et al. [3] investigated memory controller policies in cache-based systems and concluded that the simple policy of immediately transitioning the DRAM chip to a lower power state when it becomes idle is superior to more sophisticated policies that try to predict the idle time of the DRAM chip. This suggests that more software approaches are needed.

Delaluz et al. [1] proposed several compiler-directed mode control techniques to cluster the data across memory units and detect module idleness for low power. They also proposed hardware-assisted approaches based on predictions of module interaccess times.

Huang et al. [4] actively reshaped the memory traffic so that the idle periods could be aggregated to achieve low power. By exploiting this longer idle time, additional energy saving could be achieved, complementing the existing techniques.

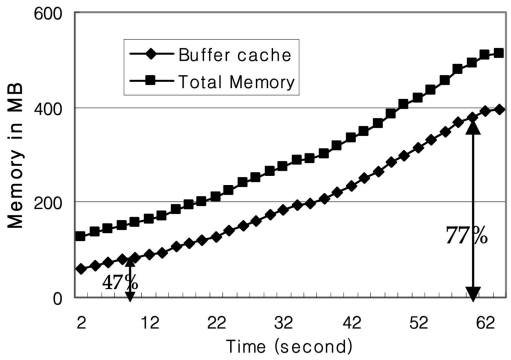


Fig. 1. Size of the buffer cache for comparing two Linux kernel sources.

Lebeck et al. [9] first studied page allocations to achieve low power consumption. The important contribution of this study is the idea that page allocation should cluster an application's pages onto a minimum number of memory chips. For this, several power-aware page allocation policies, including random and sequential first-touch placement, were evaluated. The authors also demonstrated the cooperation between the hardware and the operating system [9].

Delaluz et al. [2] proposed a simple scheduler-based policy. They utilized information on which memory units were actually being used so that the same set of memory units could be used the next time the process was scheduled.

PAVM [5] successfully combined the concepts of scheduling and page allocation. Huang et al. [6] modified PAVM to cooperate with the hardware controller so that more hints from the operating system could be exploited by the controller to achieve low power consumption.

PAVM [5] is a power management scheme for the main memory at the level of the operating system. It tracks which process uses which rank and a rank is defined as an active rank of process p if and only if at least one page from the rank is mapped into the address space of p. This active rank of process p is referred to as αp. The basic idea is to turn on only the active ranks of p and turn off all other inactive ranks when p is being scheduled. PAVM also aggregates such mapped pages into a few ranks to minimize the number of active ranks.

This scheme does not deal with memory access by the kernel space since it only considers pages mapped to the user address space. The kernel accesses the memory for various reasons, such as code fetching, manipulating kernel data structures, and copying data from the kernel space to the user space, especially for data read from the disk. These memory access events are ignored in PAVM and, thus, its performance is limited.

3 Motivation

3.1 Buffer Cache Size

Modern operating systems adopt a buffer cache. Although the buffer cache is related to the disk [7], [11], it is also related to the memory due to its size. The buffer cache usually occupies a huge area in the main memory. The simple experiment presented in Fig. 1 shows that a heavy

TABLE 1
Memory Usage in a Linux Machine (Unit: MB)

Workload	Memory used	Buffer cache	Ratio
1) Immediately after boot-up, run several X-terms	196	84	42%
2) Additionally run Opera	265	117	44%
3) Additionally run Open Office Writer	335	176	52%
4) Additionally run Open Office Spreadsheet	346	185	53%

I/O-bound job increases the buffer cache size. When the "diff" program is executed to compare two kernel source-code trees after boot-up under the Linux OS, the buffer cache eventually occupies up to 77 percent of the total 512 MB of memory.

Note that, even immediately after boot-up, the OS allocates 47 percent of the used memory to the buffer cache. This suggests that the buffer cache accounts for a significant amount of memory, even under common user workloads. We could verify this using a simple experiment with a common user workload, as shown in Table 1. This experiment was conducted on a Linux machine and the workload consisted of several X-term programs, a Web browser, and some office programs. For most cases tested, approximately half of the main memory was used for the buffer cache. Although a real workload may lead to different results, it is evident that the buffer cache should not be ignored when considering memory energy issues. Similar results were observed for the Windows OS, as shown in Table 2. These simple experiments show that the buffer cache accounts for a huge memory space under common user workloads.

TABLE 2
Memory Usage in a Windows Machine (Unit: KB)

Workload	Total memory	System cache	Ratio
1) Immediately after boot-up, 169 MB used as system cache	522988	169664	32%
2) After a small amount of use, 174 MB used as system cache	522988	174752	33%
3) During copying of large files, 379 MB used as system cache	522988	379688	72%
4) After copying, the system cache still lingers	522988	381116	73%

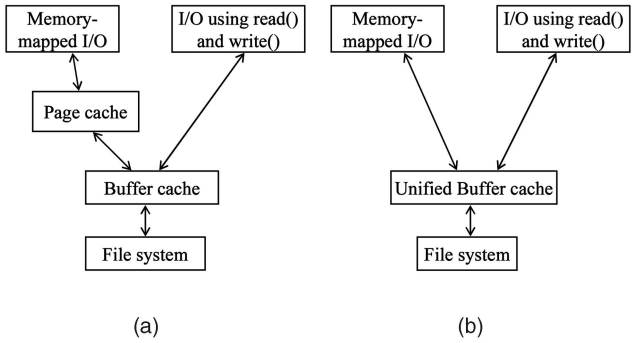


Fig. 2. (a) Page cache and (b) unified cache.

3.2 Page Cache and Unified Cache

Disk blocks can be accessed either through normal file operations or by memory-mapped I/O. A caching scheme for the former case involves a traditional buffer cache, while the latter is achieved using a page cache. If these two caching buffers exist independently, as shown in Fig. 2a, consistency issues arise. For this reason, many modern operating systems such as Linux adopt a unified cache, as shown in Fig. 2b.

Since PAVM considers memory references only from the user space, it covers only a limited part of file accesses that are serviced by the page cache through memory-mapped I/O. Moreover, this coverage is gone as soon as the process that owns the files is terminated. Our scheme is based on a unified buffer cache and, thus, it covers all of the memory accesses caused by file accesses. Also, the logical structure of buffer cache, such as hierachies, does not matter in PABC because PABC is about how the buffers are placed in the memory modules.

4 Power-Aware Buffer Cache

4.1 Introduction

In this section, we explain our new scheme, PABC, in detail. In Section 4.2, basic concepts such as the rank set, the active set, and the system pages are defined. The way PABC deals with the physical pages is also described here. In Section 4.3, the meanings and implication of the size of the system rank set are discussed. The basic ideas behind PABC are then given in Section 4.4 and the way PABC works is illustrated as well. A detailed implementation is given in Section 4.5. This subsection explains how the concepts are implemented in Linux, presenting required data structures and system issues, like page sharing and DMA. In Section 4.6, two possible policies regarding the rank set expansion are introduced along with their algorithms. In Section 4.7, we introduce a metric to evaluate and analyze the two policies. A migration scheme and its implementation are presented in Section 4.8.

4.2 PABC Basics

We classify all physical pages into three categories or page set—A_p, B_i, and Σ. A_p represents the pages of process p. This includes anonymous pages and the page tables of process p. B_i represents the buffers for the file of inode i. Σ, which we call *the system pages*, represents the physical pages

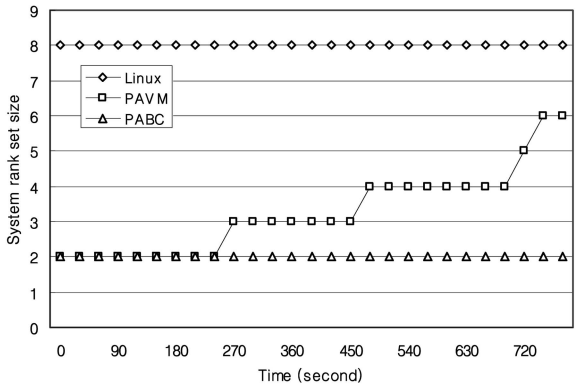


Fig. 3. Size of the system rank set for compiling the Linux kernel.

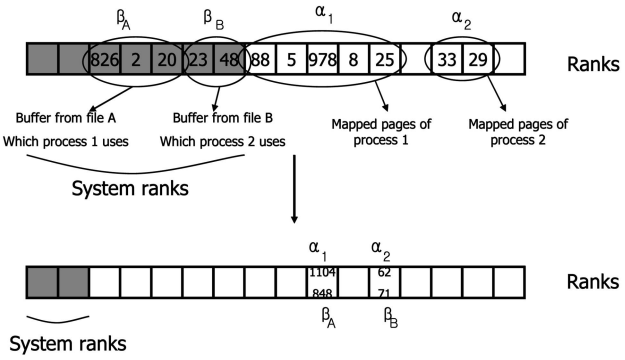


Fig. 4. Diagram of how the PABC scheme works.

that do not belong to any A_p or B_i, but contain kernel code, kernel data structures, etc.

The *rank set, denoted as* α_p, β_i, *and* S, is defined as the collection of ranks that hold any page of each category or the corresponding page set. For instance, rank set S represents the ranks that the system pages, Σ, span. Note that there is only one system rank set S, unlike the other rank sets of α_p and β_i. The system rank set S is also unique in that it always stays turned on unless the system is idle, while the other rank sets are turned on only when it is necessary. Thus, the size of the system rank set should be minimized.

We extended the kernel memory allocator to take the *rank set* as an additional parameter. This parameter allows the allocator to first try to allocate the free page from a given rank set. The kernel requests a new page with its appropriate rank set. Therefore, the pages for each rank set can be aggregated during the allocation time. When it is necessary, the allocator can expand the rank set by adding a new rank to which the allocated page belongs. How and when the rank set is expanded is explained in Section 4.6.

Using these rank sets, *the active set* and its size are defined as follows: The active set represents the ranks that should be turned on while a process is running. This active set includes α_p, β_i, and S because these are the potential ranks for memory reference.

$$Active\ set\ size = \left| S \bigcup \alpha_p \bigcup \left(\bigcup_{i=inode} \beta_i \right) \right|. \qquad (1)$$

$(\bigcup_{i=inode} \beta_i)$ is a union of rank sets of the buffers that process p uses. At a context switch, the active set for the next process is turned on and the other ranks are turned off.

In PAVM, the page set for a process contains mmap()ed pages which are mostly dynamic libraries (DLLs). This means that α_p deals with the DLLs in the light of a process. However, α_p in this paper excludes mmap()ed pages but includes anonymous pages and page tables. Since the mmap()ed pages for dynamic libraries (DLLs) have already been studied for PAVM, we focus on the buffer cache and, thus, we let β_i contain the mmap()ed pages. We deal with the DLLs in the light of a file.

4.3 Size of the System Rank Set

In PAVM, only α_p and its dynamic libraries are considered. Therefore, there could be memory references from the kernel space that were not considered. This limitation was recognized [6] and it was verified that memory access from the kernel is nonnegligible. Huang et al. [6] remedied this by aggregating pages not considered onto the first rank of the memory and by always keeping this rank active. This corresponds to the system rank set S in PABC. However, in PAVM, the system ranks contain the buffer pages B_i. Although the authors mentioned that a single rank seemed to be enough, this is not true because of the large size of the buffer cache.

Fig. 3 shows how many system ranks are needed to compile the Linux kernel. The machine has eight ranks and 256 MB of memory. "Linux" is an ordinary Linux operating system and it always turns on all eight ranks. While PABC limits the size of the system rank set to 2, the size of the system rank set increases in PAVM, mainly due to the increase in the buffer cache.

4.4 Basic Concept

The first idea is to separate the buffers from the system ranks, introducing β_i, so that the size of the system rank set can be minimized. This separation reduces the number of total active ranks because unnecessary buffers are assigned to idle ranks. Since the operating system knows which files are used by the current process, only the necessary rank sets of β_i are turned on. Since a process normally uses only a small portion of the whole buffer cache, only a few ranks need to be turned on.

The second idea is to try to locate the pages of a process and its buffers onto the same rank to minimize the active set size. For this purpose, α_p is referenced as a hint to decide the initial rank of β_i when process p causes the initial allocation for α_i.

For convenience and some technical reasons, the system rank set grows from the first rank to the next one, as shown in Fig. 4. The other rank sets are placed on arbitrary locations. Fig. 4 shows the entire memory as an array of ranks. Each rectangle is a rank and the number in it represents the number of pages allocated from a given rank. Rank sets such as α_1, α_2, β_A, and β_B are indicated by ellipses, while the system rank set is shadowed. This example shows how rank sets are aggregated and collocated. For example, the size of α_1 is reduced to 1 from 5 by

aggregating its pages into one rank. Also, α_1 and β_A are collocated onto a single rank. As a result, the size of the active rank set becomes smaller.

The third idea is to migrate pages. When active pages are scattered over many ranks, some of them can be migrated to shrink the active rank set. This process may induce significant overheads due to the copying of pages and, thus, the decision about when to migrate which pages to which rank should be made with discretion. This issue is discussed in Section 4.8.

4.5 Implementation

The PABC scheme is implemented on Linux Version 2.6.10 as follows:

```
Struct rankset {
        spinlock_t              ranks_lock;
        struct ranklist         ranks;
        struct rank_account     account;
        int                     ranks_type;
        int                     ranks_user;
        struct rankset          *reserve;
}
```

The *ranks* field is a list of ranks. The *account* field is used to account for the pages. This is a list of the same length as the ranks field and counts how many pages are allocated. These two lists together track how many pages are in use in which rank.

The kernel memory allocator tries the ranks in the *ranks* field in order. Thus, a preceding element of the list has priority over the following element in the allocation. This priority naturally enforces compaction of the rank set since the pages are allocated and aggregated onto the preceding rank as long as this has free pages. The pointer to the given rank set is left in the page descriptor if the allocation succeeds so that the rank set can be referenced when the page is freed.

The *reserve* field is used to achieve collocation of α_p and its buffers. When a file is initially opened, when no buffer has yet been allocated, this field is set to point to α_p for the process that opened it. Later, the initial rank for the buffer is taken from this α_p.

Each instance of this structure falls into one of three categories, i.e., α, β, or S, by the *ranks_type* field and is protected by the *ranks_lock* field.

The system rank set, S, is initialized as the first two ranks at boot-up initialization due to DMA. Since DMA references physical pages without CPU intervention, such pages should be allocated to the system rank set and rank 0 should be included in the system rank set because the ISA-DMA controller uses less than 16 MB of the physical memory. For convenience of implementation, we set the second rank, rank 1, as the main system rank. Thus, rank 0 is primarily used for DMA and rank 1 is the main system rank. These two ranks are pinned so that they cannot be removed from the system rank set.

For rank sets of type α or β, the initial rank from which the first page is allocated is very important. Because it is common that almost all pages of a rank set are allocated from the same rank, once this initial rank is decided, subsequent page allocation is carried out from this rank. Thus, the initial rank should be determined with discretion. This is discussed in Section 4.6.

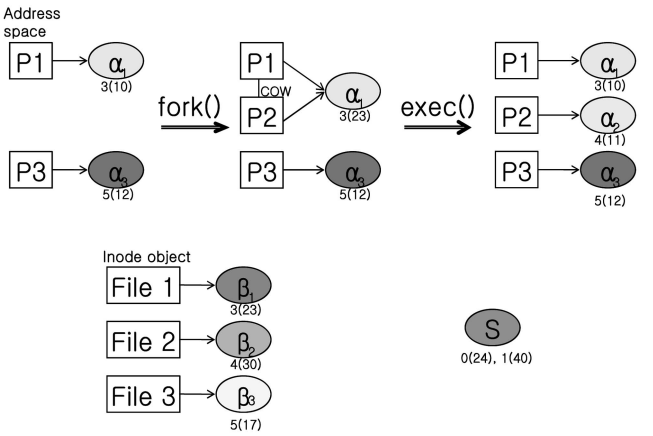


Fig. 5. Relationship between the rank sets and kernel data structures.

Fig. 5 shows how the rank sets are related to the other kernel data structures. The rectangles represent address spaces and the ellipses represent rank sets. The numbers below the ellipses show how many pages in each rank are used. For instance, 3(10) indicates the 10 pages in the third rank are used for this rank set. Since α_1 is related to the P1 address space, then these 10 pages are mapped to the P1 address space. Similarly, β_1 manages 23 pages in rank 3 to cache file 1. Since α_p manages the physical pages of process p, it is associated with the address space of p. While β_i is associated with only one inode object, several address spaces that share some pages may share α_p. Page sharing is frequent when a new process is forked since most physical pages are shared by the copy on write (COW) mechanism. Therefore, α_p is inherited across the fork system call and a new rank set is only created and allocated by the creation of a new address space through the *exec* system call.

The *ranks_user* field of the rank set structure is meaningful only when the structure is of type α. This field counts the number of address spaces that use the rank set so that the rank set is freed only when this field equals zero. A rank set of type β is allocated when the first buffer is allocated for the file and is freed when all buffers for the file are freed. The system rank set, S, is allocated at system boot-up and is freed on system shutdown.

Fig. 6 shows the changes in memory usage when a new process is created as in Fig. 5. The first two ranks are used by the system and rank 3 is used by process P1. After P1 creates a child process P2, the memory for P2 is allocated on α_1, as in Fig. 6b. Then, P2 discards its address space by an exec() system call and some pages of α_1 are freed. As a new address space that does not share any physical pages with any other address space is created, a new rank set, α_2, is allocated for P2 and rank 4 is chosen as its initial rank.

4.6 Rank Set Expansion Policy

A rank set is expanded when the pages that should be allocated exceed its capacity. This increases the rank set size, which in turn leads to energy consumption. Thus, rank set expansion should be allowed by a suitably prudent

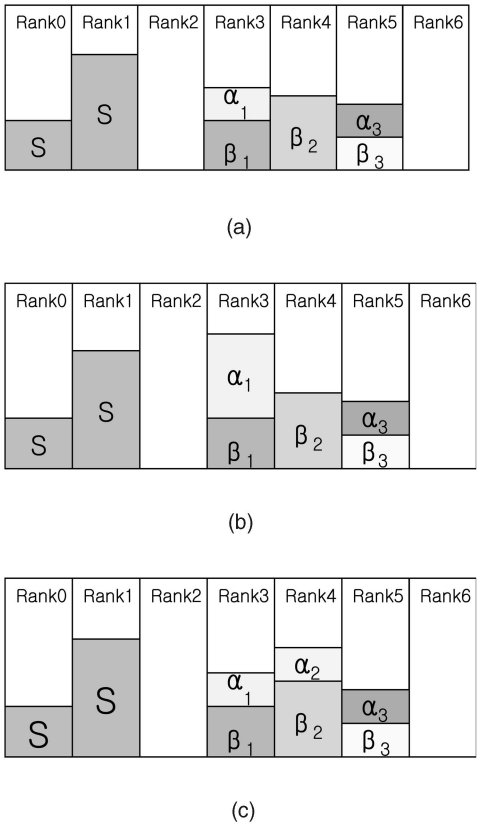


Fig. 6. Actual memory usage through the rank sets. (a) Example of memory layout. (b) After executing fork(). (c) After executing exec().

policy. On the contrary, the rank set may shrink when all of the pages allocated to a specific rank are freed.

To expand the rank set, a decision is needed on which rank to choose as the initial rank or new rank to be added. PABC simply chooses the rank with the most free pages. However, for the initial element of β, PABC chooses a rank from the rank set that belongs to the relevant process to achieve collocation, as explained in Section 4.5.

In addition, a decision is needed on when to expand the rank set. For this, we introduce a new simple policy for rank set expansion, termed PABC-advanced, and refer to the old policy as PABC-basic. Fig. 7 gives an approximate depiction of these two policies. PABC-basic tries to allocate the free pages from a given rank set and, if it fails, simply expands the rank set. However, PABC-advanced tries to reclaim pages from a given rank set without disk I/O. Since disk I/O may possibly cancel the benefits of the PABC scheme, it prefers rank set expansion to disk I/O.

Although this early PABC-advanced scheme has worked well under light workloads, it has two problems. First, it suffers from reclamation of active pages when ranks have few inactive pages. We refer to a rank as a *busy* rank when more than 80 percent of the pages are active. This problem of reclaiming active pages in busy ranks, despite the presence of many inactive pages in other ranks, degraded the overall performance.

To remedy this problem, the early PABC-advanced scheme was revised to utilize the concept of *active* and *inactive* pages in Linux. Since PABC is implemented using the zoned memory allocator of Linux and Linux maintains

```
Boolean  try_to_get_free_page_basic(rankset) {
        If (allocation succeeds from any rank in the rank set) {
                Allocate;
                Return true;
        }
        /* expand the rank set */
        allocable_rank = get_new_rank_to_add_basic(rankset);
        if (allocable_rank == null) return false;
        allocate from allocable_rank;
        add allocable_rank to rankset;
        return true;
}
get_new_rank_to_add_basic(rankset) {
        rank =
        find_rank_of_most_free_pages_not_belong_to_rankset();
        return rank;
}
```

(a)

```
Boolean try_to_get_free_page_advanced(rankset) {
        If (allocation succeeds from any rank in the rank set) {
                Allocate;
                Return true;
        }
        if (not(all ranks in this rankset are busy,
        i.e. all ranks have at least 80% active pages))
        Try_reclaim_pages_without_diskIO_from(rankset);

        /* expand the rank set */
        allocable_rank =
                get_new_rank_to_add_advanced(rankset);
        if (allocable_rank == null) return false;
        allocate from allocable_rank;
        add allocable_rank to rankset;
        return true;
}
get_new_rank_to_add_advanced(rankset) {
        rank =
        find rank of max value of free_pages+nr_inactive/2
                which does not belong to rankset;
        return rank;
}
```

(b)

Fig. 7. Pseudocode for the memory allocator. (a) PABC-basic. (b) PABC-advanced.

active and inactive lists for each zone, it can be easily identified whether a given rank is busy or not. Therefore, the early PABC-advanced was revised to allow rank set expansion if all ranks in the rank set are busy.

Although this modification successfully stabilized the performance, a second problem was revealed. During system operation, an imbalance in the number of inactive pages was found. Since a rank with the most free pages is chosen as the new rank to be added, ranks with many inactive pages tended to be excluded and ranks with many active pages tended to be included as candidates. Thus,

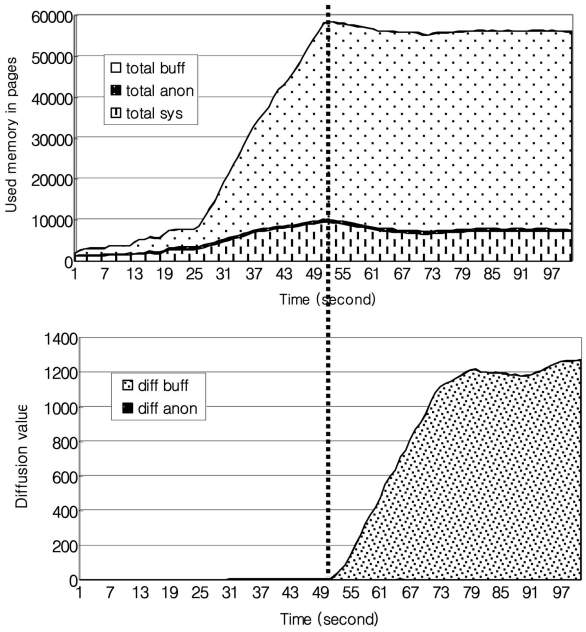


Fig. 8. Increase in diffusion value as the memory runs out.

ranks with many active pages tended to be reclaimed frequently and in turn tended to have many free pages.

To remedy this, the early PABC-advanced scheme was finally revised to use "free_pages+nr_inactive/2" instead of just "free_pages" to find the emptiest rank. This views inactive pages as potential free pages. Since inactive pages have a long LRU distance, it is reasonable to treat a rank with many inactive pages as a reasonably empty rank. This contributes to better balance among the ranks and prevents the appearance of busy ranks.

These two modifications successfully solved the problems and showed stable performance and good balance among the ranks. The modifications also play a major role in preserving the hit ratio because they ultimately enable PABC-advanced to preserve active pages and to reclaim inactive pages fairly among the ranks. This is discussed in Section 5.4.

4.7 Compaction

If the size of a rank set is minimal, which is usually 1, we say that the rank set is compact. We refer to the difference between a rank set size and the minimal set size as the *diffusion value*, for brevity, to analyze and evaluate the scattering, or diffusing effect of the rank set. As the memory runs out, each rank set may expand and the size of rank set increases. Fig. 8 shows the increase in diffusion values as the memory runs out on running the *diff* program to compare two Linux kernel source codes after boot-up. PABC-basic was used in this experiment.

Diff_anon is the total diffusion value for all α rank sets and Diff_buff is the total diffusion value for all β rank sets. Total_anon, Total_buff, and Total_sys are the number of pages used for α (process memory), β (buffers), and S (system memory), respectively. As shown in Fig. 8, PABC-basic yields perfect compaction of zero diffusion values at the start. However, the diffusion value increases sharply as soon as the memory runs out. A high diffusion value indicates greater energy consumption. This is one limitation of PABC-basic. Since PABC-advanced suppresses the expansion of rank sets, it leads to low diffusion values and achieves good compaction. Thus, PABC-advanced overcomes the limitation of PABC-basic and this is shown in Section 5.

4.8 Migration

Experiments revealed that the size of buffers allocated for a single file is quite small in the *Light* workload scenario in Table 3. The buffers for most files occupy less than 10 pages. This is salient in the *Light* workload and these small buffers prevent PABC-advanced from reducing the active rank size. It is desirable to collect such small buffers into a few ranks and, thus, a buffer migration mechanism was introduced as a kernel thread.

Because buffer migration incurs overhead, we carefully designed a buffer migration daemon, *bmigrated*, to be effective at low overhead. This daemon wakes up at every 3 s and checks whether the system is idle. If it is not, it immediately yields CPU and sleeps. If the system is idle, it picks up the best of the candidate processes and migrates its buffers to shrink the size of its active rank set. This daemon processes one candidate at a time to avoid creating large memory traffic.

The list of candidate processes is maintained as shown in Fig. 9. The active set size of 3 is minimal in our implementation, which includes only system rank sets of size 2 and 1. Therefore, active sets of size 4 or above are candidates for migration. Thus, a process is added to the list when its active set size reaches 4. When the daemon wakes

TABLE 3
Workload

Application	Interval	Light	Poweruser	Multimedia	Description
X+GNOME	Continuous	X	X	X	Runs X server using the default GNOME desktop environment
Firefox	15 s	X	X		Retrieves and displays web pages from randomly pre-generated URL
Mplayer	Continuous	X	X		Plays a stream of mp3 files
Text editing	60 s	X	X		Modifies a tex file, runs latex, dvipdf, and display it in gpdf
Gcc	Continuous		X		Compiles Linux-2.6.10 kernel and kernel modules; Make clean and make;
Mplayer	Continuous			X	Plays an MPEG4-encoded movie in full-screen mode

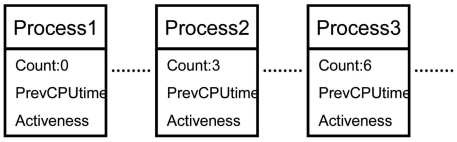


Fig. 9. Migration candidate list.

up and the system is idle, it scans the list to find the best candidate and updates its fields. These fields are explained below. In addition, if the size of the active rank set of the candidate is less than 4, it removes the process from the list. Once the best candidate is chosen, the daemon shrinks the active set size by migrating its buffers and removes the process from the list.

The ideal candidate for migration is a CPU-bound job with low migration cost. Because our global goal is to minimize subsequent energy consumption, CPU utilization by a process is taken into account. Although this formula does not identify the exact energy consumption, it yields a good approximation as a modeled value, rtime. The definition of rtime and details of this formula are presented in Section 5.1. The exact energy consumption depends on the underlying memory technology:

$$Energy\ consumption = \sum_{Each\ time\ slice} T(p) \times\ active\ set\ size. \tag{2}$$

$T(p)$ denotes the number of ticks and can be understood as the CPU utilization of process p. From this formula, it is evident that it is important to reduce the active set size of a process with high CPU utilization. Therefore, our migration scheme mainly focuses on the detection of these CPU-bound processes.

The migration cost is approximately the ratio of the number of pages to migrate,to the number of ranks that will be turned off after migration. The number of pages to migrate is the obvious cost and the benefit of this migration is how many ranks can be turned off, hence yielding the migration cost.

To pick the best candidate, we classify processes into three categories: short-lived processes, long-lived processes with low CPU utilization, such as daemons or user interactive programs, and long-lived processes with high CPU utilization, such as playing MP3 files. Processes in the third category represent good candidates for migration because of their long duration and high CPU utilization.

As shown in Fig. 9, the *bmigrated* daemon uses three fields to differentiate processes into the three categories. When the daemon scans the migration candidate list, it increments the count value from 1 to 6. If a process lives long enough, the *count* value reaches 6. Once candidates have reached a count value of 6, they are evaluated on CPU utilization rather than lifetime. The upper limit for the count value is used to confer a disadvantage to short-lived processes rather than to prioritize long-lived processes since the lifetime of the latter is not as critical as for short-lived processes.

To track CPU utilization at a low overhead, the daemon maintains the *prevCPUtime* and *activeness* fields. Since Linux tracks how many ticks each process spends in the CPUtime

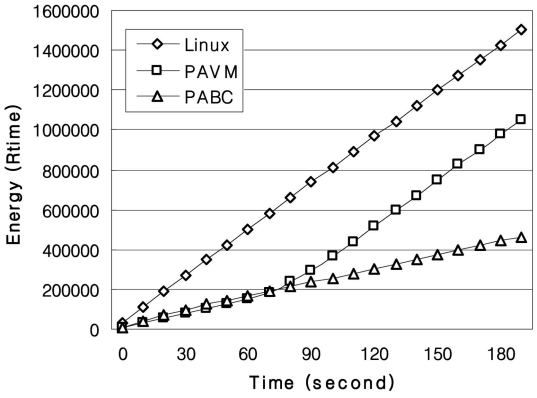


Fig. 10. Power consumption to run the diff program.

data structure, we use the difference between the previous and current CPUtime as the CPU utilization metric and use it in a history-sensitive formula as follows: Once the activeness is calculated, the daemon puts current CPU time in the prevCPUtime field for the next calculation:

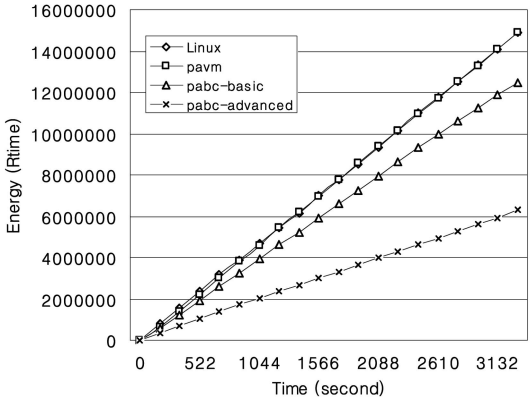
$$Activeness_n = \frac{1}{2}(currentCPUtime - prevCPUtime) + \frac{1}{2}(activeness_{n-1}). \tag{3}$$

As a final metric, we define the candidate value (CV) as follows:

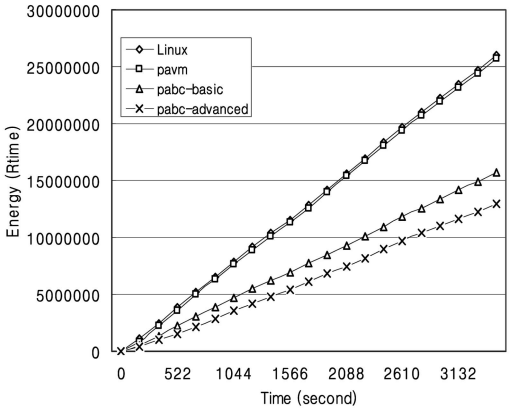
$$CV = \frac{1}{migration\ cost} \times activeness \times count. \tag{4}$$

This candidate value depends on three variables. A high value for the migration cost prevents migration. The activeness value gives preference to CPU-bound processes and the count value confers disadvantages on short-lived processes. Therefore, the *bmigrated* daemon prefers long-lived and active processes with low migration cost. It chooses the candidate with the greatest CV.

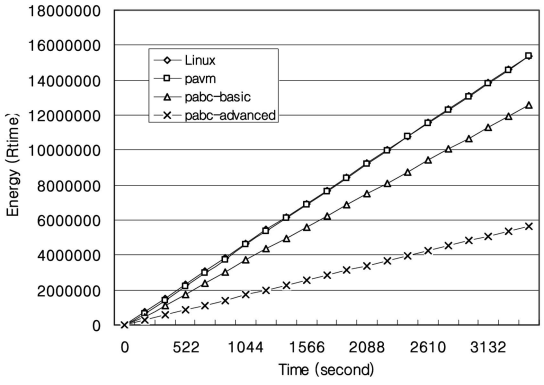
A case can arise for which no candidate can be chosen for certain reasons. If no long-lived process exists or the migration cost is too high, no migration occurs. In addition, if the rank to migrate into does not have enough free pages, the daemon does not choose such a process. Buffers for shared files are not migrated since they may be migrated continuously between two processes. Moreover, during the *Multimedia* workload experiment shown in Table 3, we found that the *bmigrated* daemon migrated a huge number of buffers, almost a whole rank, when it was the only candidate to migrate. Since page migration itself consumes energy and is an expensive operation, the number of pages to migrate should be limited. Thus, our scheme ignores the candidate if the number of pages to migrate is greater than 100. Even though we could not afford to explore the entire search space, this value proved to be the most effective in the cases tested.



(a)



(b)



(c)

Fig. 11. Energy consumption under the workloads outlined in Table 3. (a) Light. (b) Poweruser. (c) Multimedia.

5 Evaluation

5.1 Experimental Set-Up and Metric

All the experiments in this section were conducted on a machine equipped with 256 MB of DDR SDRAM and a Pentium 4 2.8-GHz CPU. A rank is defined as a chunk of memory of 32 MB, so there are a total of eight ranks and each rank has up to 8,192 pages. We implemented PABC on a Linux 2.6.10.

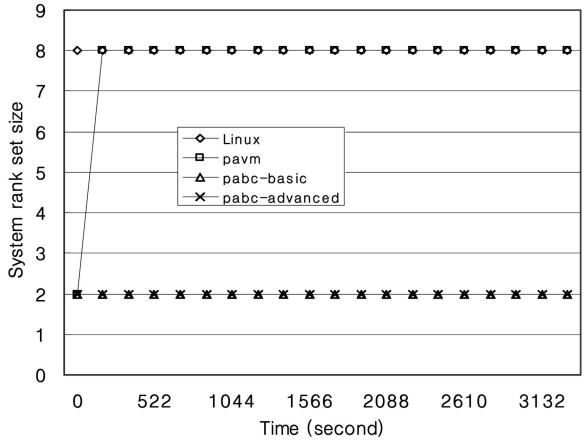


Fig. 12. Limiting the system rank size.

Since we need a metric unit for measurements, we define *rtime*, rank time, as the amount of energy consumed by one active rank for one tick (1/1000 s in this experiment). To measure the energy consumed, each process accumulates the rtime value for the ticks it spends for a given time quantum. Because each process activates as many ranks as the size of its active set, this can be measured as follows:

$$Rtime \mathrel{+}= ticks\ the\ process\ spends \times active\ set\ size. \quad (5)$$

5.2 Workload

Almost the same workloads as those used in PAVM [5] were adopted, as summarized in Table 3. The *Light* workload is a suite of jobs that represents ordinary computer usage. It includes playing mp3 files, Web surfing, and using LaTeX. The *Poweruser* workload represents a more advanced user, such as a computer programmer, and includes kernel compilation in addition to the *Light* workload. The *Multimedia* workload mainly includes playing a movie in full-screen mode. This reflects the trends for multimedia computing.

All measurements were made after cold boot-up. The results were highly reproducible and consistent. Migration was disabled for these experiments. This is discussed in Section 5.5.

5.3 Energy Saving

Among many workloads tested, the *diff* program is an intensive IO-bound job and, thus, it clearly shows how effectively each scheme handles the buffer cache. Fig. 10 shows the energy consumption required to run the *diff* program, which compares two Linux kernel source trees. The PAVM and PABC-basic schemes are compared, while unmodified Linux was measured as a yardstick. PAVM saves power for 1 minute after boot-up, but soon begins to lose efficiency, following the same slope as Linux, while PABC-basic continues to save energy for the whole time measured. The limitation of PAVM can be explained by the fact that the system rank set grows as the size of buffer cache increases and the system rank set spans all of the ranks soon after boot-up.

The results for running the workloads outlined in Table 3 are shown in Fig. 11. Overall trends show that PABC-advanced is the best for all cases tested, followed by

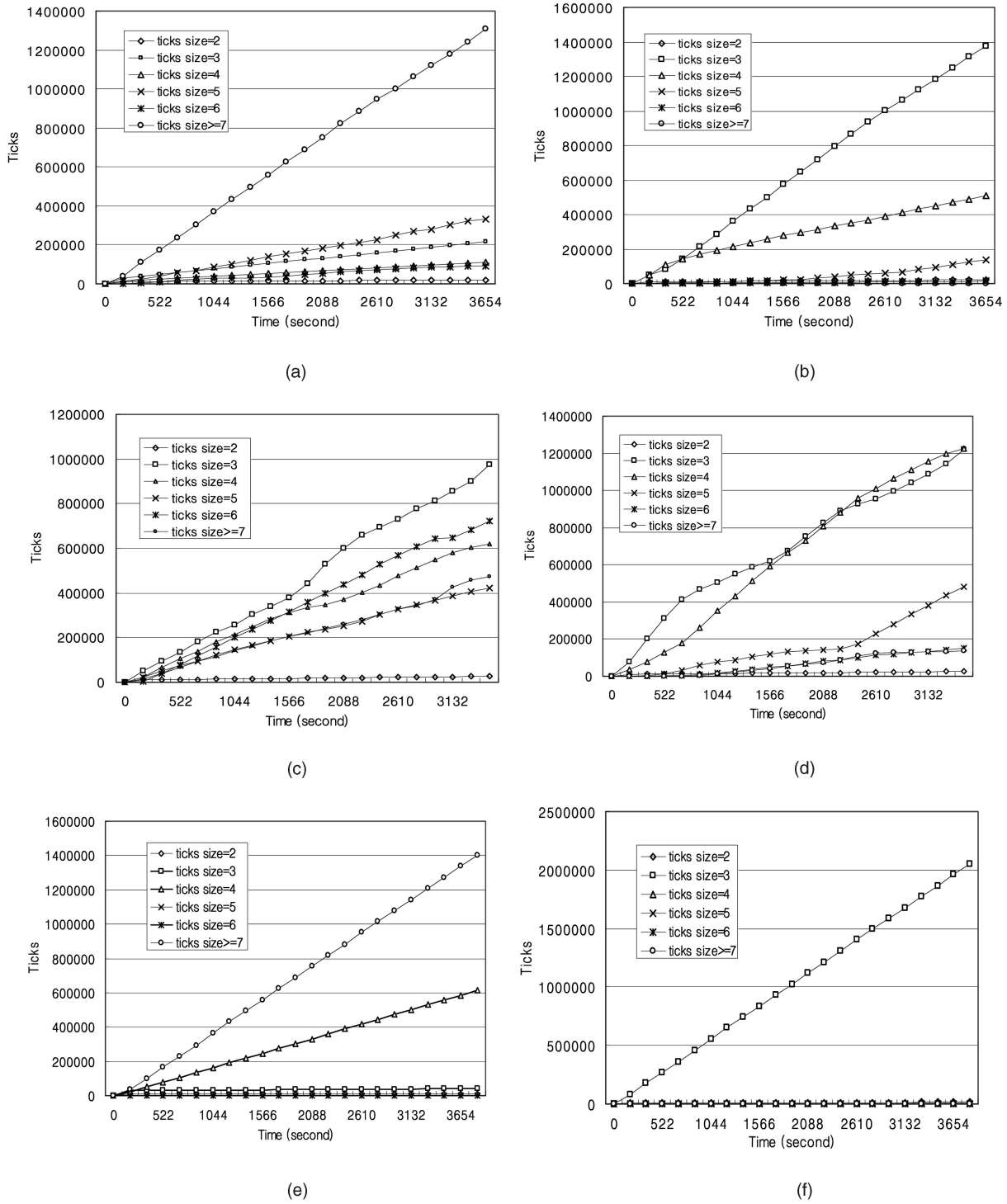


Fig. 13. Breakdown of power consumption for the rank sizes. (a) PABC-basic/Light. (b) PABC-advanced/Light. (c) PABC-basic/Poweruser. (d) PABC-advanced/Poweruser. (e) PABC-basic/Multimedia. (f) PABC-advanced/Multimedia.

PABC-basic, while PAVM shows a rather small energy saving. PABC-basic reduces the power by 20-40 percent, while the power saving for PABC-advanced is 50-63 percent.

We also measured the system rank size for each scheme, as shown in Fig. 12. All the workloads produced the same result in Fig. 12. This shows that the size of the system rank sets for two PABC schemes is limited to 2, while Linux and PAVM increase the size to a maximum value. This explains how the PABC schemes save energy.

Since the only difference between PABC-advanced and PABC-basic is the way in which the degree of diffusion is controlled, this should dictate the gap between them. To reveal the reason for the performance gap between PABC-basic and PABC-advanced, we break down the energy consumption in Fig. 13 to show how many ticks are spent for each rank set size. Although not shown here, we observed that Linux and PAVM spend most of the time ticks using all of the ranks.

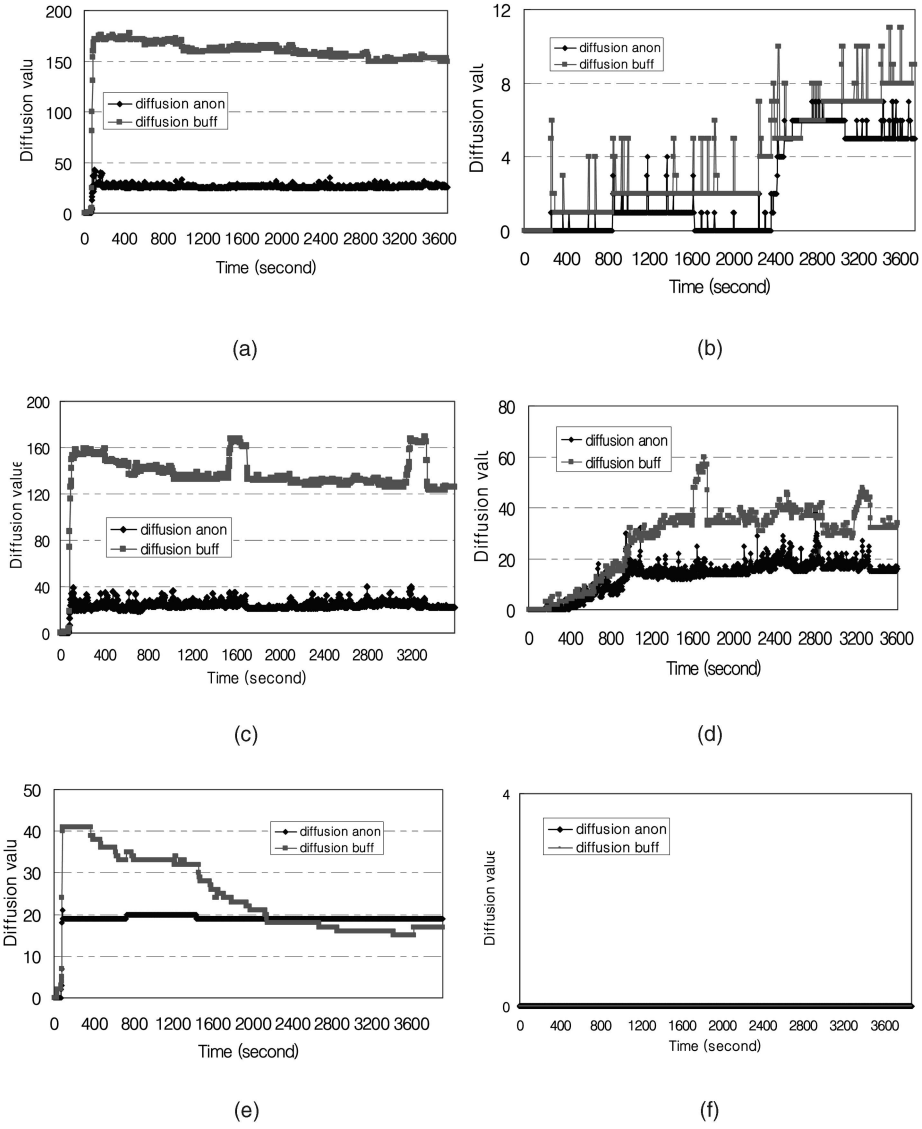


Fig. 14. Diffusion values. (a) PABC-basic/Light. (b) PABC-advanced/Light. (c) PABC-basic/Poweruser. (d) PABC-advanced/Poweruser. (e) PABC-basic/Multimedia. (f) PABC-advanced/Multimedia.

PABC-basic spends most ticks on rank sets of size 7 or above, as shown in Fig. 13a and Fig. 13e. Considering the limited number of system ranks, this unsatisfactory performance is due to its lack of compaction policy for rank sets. Unlike PABC-basic, PABC-advanced spends most ticks over a smaller number of ranks owing to its aggressive compaction policy.

Note that these experimental results depend on the number of total ranks. We chose a rather small number of ranks, eight, for the sake of experiments and analysis. However, we can expect that a smaller rank size would produce better performance owing to finer control of the ranks. In addition, an increase in the number of ranks due to greater memory would allow more ranks to be turned off so that PABC would show better performance in its percentile metrics. This is because the active set size remains the same, regardless of memory size.

5.4 Compaction

Fig. 14 shows the diffusion values: "diffusion anon" shows the sum of diffusion values for α rank sets and "diffusion buff" shows the sum of diffusion values for β rank sets. The diffusion values for all workloads are significantly reduced when the compaction scheme is used. The compaction strategy was especially effective for the *Light* and *Multimedia* workloads. A small diffusion value was achieved by compaction for the *Light* workload. For the *Poweruser* workload, the diffusion value in Fig. 14d is greater than that in Fig. 14b because of the additional kernel compiling. This is expected because kernel compiling aggressively invokes file I/O operations. However, it is still smaller than the value in Fig. 14c due to the compaction.

It is interesting that the diffusion values in Fig. 14c are smaller than those in Fig. 14a, despite the kernel compiling. We believe that the frequent page reclaiming involved in kernel compiling possibly provided some free pages, which could prevent the expansion of the rank sets. For the *Multimedia* workload, perfect compaction was achieved in

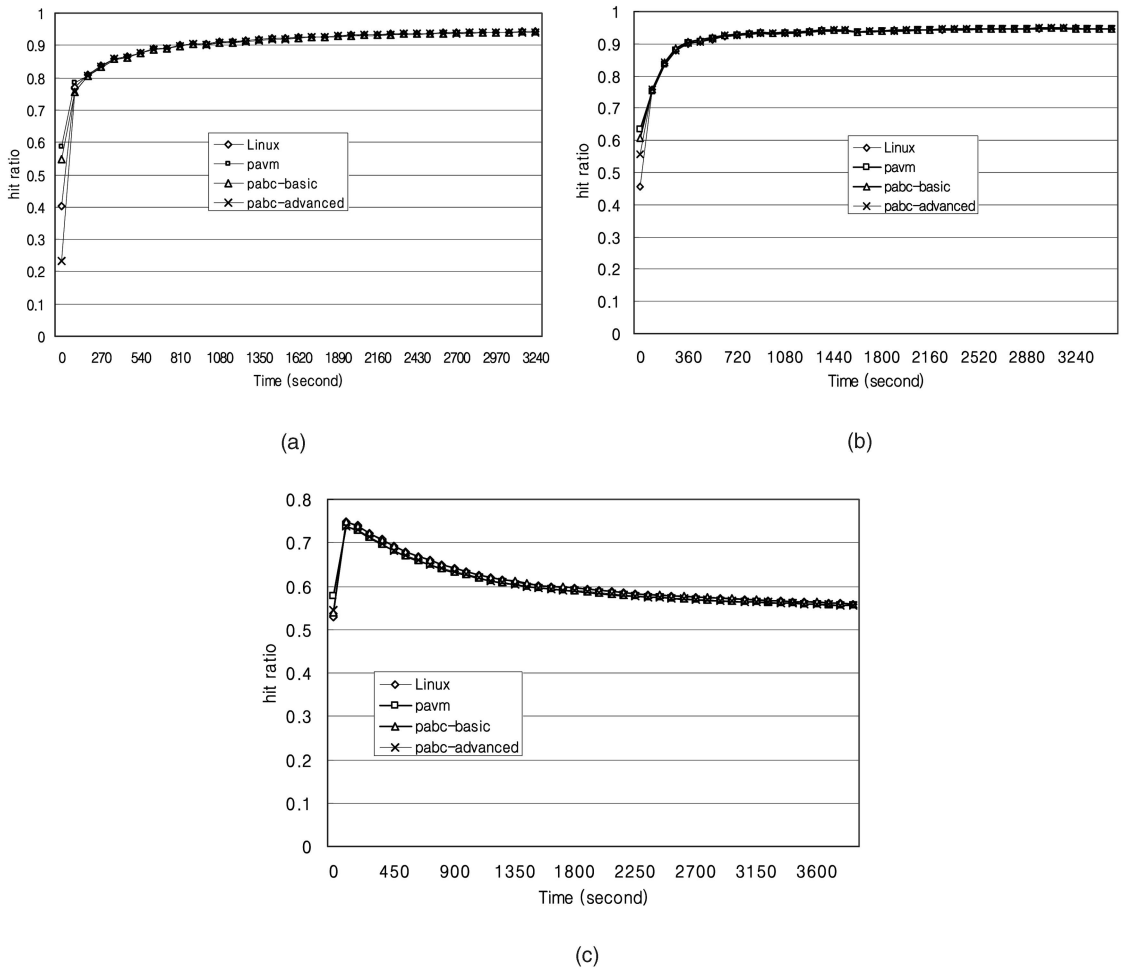


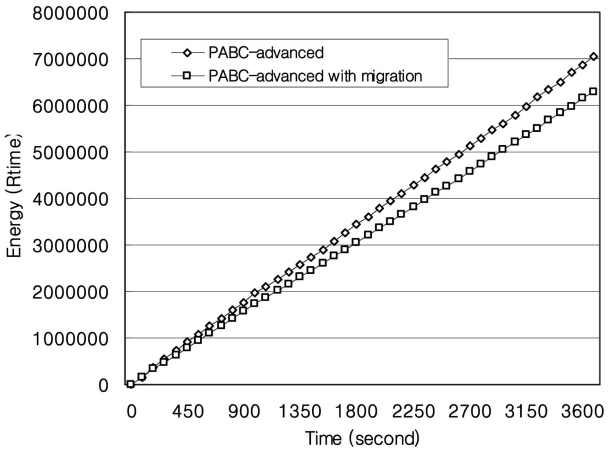
Fig. 15. Impact on the hit ratio. (a) Light workload. (b) Poweruser workload. (c) Multimedia workload.

Fig. 14f. This is due to the sequential access of the workload. Since this workload retains a small number of active pages and only Mplayer is executed, perfect compaction could be achieved.

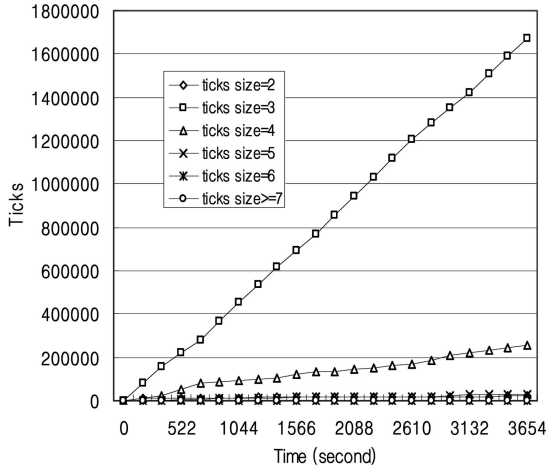
The compaction scheme may lead to a lower cache hit ratio since it reclaims buffers at an earlier time compared to the original buffer replacement scheme. Since a candidate is chosen from a given rank set rather than from the whole cache, this is different from the one chosen by the LRU policy. The hit ratios measured under the compaction scheme are shown in Fig. 15. Since the read-ahead logic of Linux significantly affects the hit ratio, this feature was turned off throughout this experiment for better observation of changes in the hit ratio. When we conducted these experiments with read-ahead logic, we observed hardly any meaningful differences. Contrary to our intuitive expectation, the impact of compaction on the hit ratio was not significant. Using read-ahead logic, the differences were even smaller.

The reason why the hit ratio is preserved is that the total cache capacity is preserved and PABC-advanced is well adjusted. Although the PABC schemes split the buffer cache into several ranks, the total cache capacity is still the same. It is clear that PABC-basic does not affect the hit ratio because it does not affect the page replacement scheme, but rather splits them into several ranks. However, PABC-advanced may cause early eviction of pages, even if free pages exist. However, this does not decrease the hit ratio unless the process causes a whole rank to be filled with its actively used buffers, i.e., a busy rank. Such a case rarely occurs since PABC-advanced balances active pages among the ranks. Even if such a busy rank appears, PABC-advanced expands its rank sets rather than reclaiming pages from the busy rank. Since PABC-advanced pursues the balancing of active pages among the ranks and suppresses the appearance of busy ranks, the hit ratio can be preserved.

PABC-advanced sometimes even shows a slightly better hit ratio, contrary to our expectations. We believe that work carried out by Kim et al. [8] can provide a clue to explain this. The authors achieved a better hit ratio than for existing replacement policies by exploiting the reference regularities of block I/O. Their new scheme, called unified buffer management (UBM), detects sequential, looping, and other references so that these regularities can be exploited for a better hit ratio. Possibly, forcing the allocation of buffers from the corresponding rank set simulates UBM to a certain degree. Since a process tends to exhibit one of such regularities, it is possible that reclaiming buffers from the β_i rank sets of a process that exhibits sequential references prevents more valuable buffers from being reclaimed. Therefore, we believe that our observation can be explained, at least partially, as a UBM effect.



(a)



(b)

Fig. 16. Effect of migration in the Light workload. (a) Energy consumption. (b) Breakdown into rank set size.

5.5 Migration

The migration daemon, *bmigrated*, rarely migrated buffers for the *Poweruser* workload because the system was too busy for the daemon to run. For the *Multimedia* workload, the daemon did not migrate any buffer because the number of buffers to migrate was huge—over 5,000 pages. Thus, the daemon only worked for the *Light* workload, as shown in Fig. 16. PABC-advanced with migration showed an additional 10 percent energy reduction compared to PABC-advanced. Fig. 16b shows how this additional energy reduction is achieved; the number of ticks spent with rank set size 4 or above is much smaller than in Fig. 13b, while the number of ticks spent with rank set size 3 is greater than in Fig. 13b. This implies that the migration scheme is working and suppresses any increase in the active set size.

5.6 Overhead

The major runtime overhead is to turn the rank sets on and off at every context switching time. This overhead, however, is hidden from application latencies since these operations can be carried out in parallel while processes are switched [5]. Other overheads, such as accounting, allocation, and freeing of the rank sets, are negligible since these are constant time operations. Although the kernel memory allocator scans the ranks in a given rank set, the rank set size is small enough to be treated as a constant time. When the rank set expands, the allocator scans the ranks to find the emptiest rank, which is a linear operation. However, the number of total ranks is still small enough. In addition, the spatial overhead of the rank set structures and the pointer to the rank set in the page descriptor is within the acceptable range: 1 MB was enough for our experiments in Section 5. Therefore, we believe that these overheads are not critical.

The overhead for migration includes operations for scanning the migration candidate list and the migration of buffers. The candidate list was quite short throughout our experiments. For the *Light* workload, the list tended to have less than five processes. Moreover, the number of buffers to be migrated was less than 10 pages for most cases tested. Since a migration policy may impose a limit on the number of pages to be migrated, this overhead can be bounded, as in our experiments. In any case, since the migration daemon works only when the system is idle, it rarely affects the other processes.

The energy used to migrate pages is not easy to directly compare since it depends on the underlying memory technology. However, since a long running process is preferred as a migration candidate and since only a few pages are usually migrated, the energy cost would be negligible compared to the energy consumption that would be incurred to keep more ranks turned on. This is why we classified the processes into three categories for the best candidate in Section 4.8. Throughout the experiments, those selected processes have shown long enough lifetimes and we believe that the energy overhead for migrating pages does not outweigh the savings.

6 Conclusion

This paper presents a scheme to conserve energy consumed by memory systems. Although this work can be considered an extension of PAVM, the observations and schemes suggested to realize intuitive ideas are much more effective than PAVM. Separation of the buffer cache from the system memory allows the suggestion of several fruitful methods for reducing the number of active memory ranks. Based on the observation that a buffer cache entry belongs to a file and the file is usually owned by a process, we present a compaction scheme that saves up to 63 percent of the energy consumed. The migration of pages allows a further reduction in the number of active ranks and achieves an additional energy saving of 4.4 percent.

Acknowledgments

This research was supported by the Ministry of Information and Communication, Korea, under the Information Technology Research Center support program supervised by the Institute of Information Technology Assessment (IITA-2005-C1090-0502-0031).

REFERENCES

[1] V. Delaluz et al., "Hardware and Software Techniques for Controlling DRAM Power Modes," *IEEE Trans. Computers,* vol. 50, no. 11, pp. 1154-1173, Nov. 2001.

[2] V. Delaluz et al., "Scheduler-Based DRAM Energy Management," *Proc. 39th Design Automation Conf.,* 2002.

[3] X. Fan, C.S. Ellis, and A.R. Lebeck, "Memory Controller Policies for DRAM Power Management," *Proc. Int'l Symp. Low Power Electronics and Design,* 2001.

[4] H. Huang, C. Lefurgy, T. Keller, and K.G. Shin, "Memory Traffic Reshaping for Energy-Efficient Memory," *Proc. Int'l Symp. Low Power Electronics and Design (ISLPED '05),* pp. 393-398, Aug. 2005.

[5] H. Huang, P. Pillai, and K.G. Shin, "Design and Implementation of Power-Aware Virtual Memory," *Proc. USENIX Ann. Technical Conf.,* pp. 57-70, 2003.

[6] H. Huang et al., "Cooperative Software-Hardware Power Management for Main Memory," *Proc. Workshop Power-Aware Computer Systems,* Dec. 2004.

[7] S. Jiang, X.N. Ding, F. Chen, E.H. Tan, and X.D. Zhang, "DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities," *Proc. Fourth USENIX Conf. File and Storage Technologies (FAST '05),* Dec. 2005.

[8] J.M. Kim, J. Choi, J. Kim, S.H. Noh, S.L. Min, Y. Cho, and C.S. Kim, "A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References," *Proc. Operating Systems Design and Implementation (OSDI),* Oct. 2000.

[9] A.R. Lebeck et al., "Power Aware Page Allocation," *Proc. Architectural Support for Programming Languages and Operating Systems,* pp. 105-116, 2000.

[10] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. Keller, "Energy Management for Commercial Servers," *Computer,* vol. 36, no. 12, pp. 39-48, Dec. 2003.

[11] Q.B. Zhu and Y.Y. Zhou, "Power Aware Storage Cache Management," *IEEE Trans. Computers,* vol. 54, no. 5, pp. 587-602, May 2005.

Min Lee received the BS and MS degrees in computer science from Yonsei University, Korea, and the Korea Advanced Institute of Science and Technology (KAIST) in 2004 and 2006, respectively. He is currently a PhD student at the Georgia Institute of Technology. He received the best paper prize of 2006 Graduation at KAIST for his PABC work and also received some prizes in the Samsung Humantech Thesis Award competition and the 25th student's thesis competition held by the Korea Information Science Society. His research interests include operating systems, virtualization, computer architecture, and artificial intelligence.

Euiseong Seo received the BS and MS degrees from the Korea Advanced Institute of Science and Technology (KAIST), Korea. He is a PhD candidate in the Computer Science Department of KAIST and is affiliated with the Computer Architecture Lab. His research is on power-aware resource management, energy-efficient real-time scheduling, and, recently, performance improvement of virtual machines. His works have been published in refereed journals and in internationally refereed conferences.

Joonwon Lee received the BS degree in computer science from Seoul National University in 1983 and the MS degree and PhD degrees from the Georgia Institute of Technology in 1990 and 1991, respectively. Since 1992, he has been a professor at the Korea Advanced Institute of Science and Technology (KAIST), Daejon, Korea. His current research interests include low power embedded systems, system software, and virtual machines.

Jin-soo Kim received the BS, MS, and PhD degrees in computer engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He was with the IBM T.J. Watson Research Center as an academic visitor from 1998 to 1999. Since 2002, he has been a faculty member in the Department of Electrical Engineering and Computer Science at the Korea Advanced Institute of Science and Technology (KAIST). Before joining KAIST, he was a senior member of the research staff at the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002. His research interests include operating systems, embedded systems, cluster computing, and storage systems. He is a member of the IEEE and the ACM.