

# Delay-Hiding Energy Management Mechanisms for DRAM

Mingsong Bi, Ran Duan, Chris Gniady  
University of Arizona  
{mbi, tomduan, gniady}@cs.arizona.edu

## Abstract

*Current trends in data-intensive applications increase the demand for larger physical memory, resulting in the memory subsystem consuming a significant portion of system's energy. Furthermore, data-intensive applications heavily rely on a large buffer cache that occupies a majority of physical memory. Subsequently, we are focusing on the power management for physical memory dedicated to the buffer cache. Several techniques have been proposed to reduce energy consumption by transitioning DRAM into low-power states. However, transitions between different power states incur delays and may affect whole system performance. We take advantage of the I/O handling routines in the OS kernel to hide the delay incurred by the memory state transition so that performance degradation is minimized while maintaining high memory energy savings. Our evaluation shows that the best of the proposed mechanisms hides almost all transition latencies while only consuming 3% more energy as compared to the existing on-demand mechanism, which can expose significant delays.*

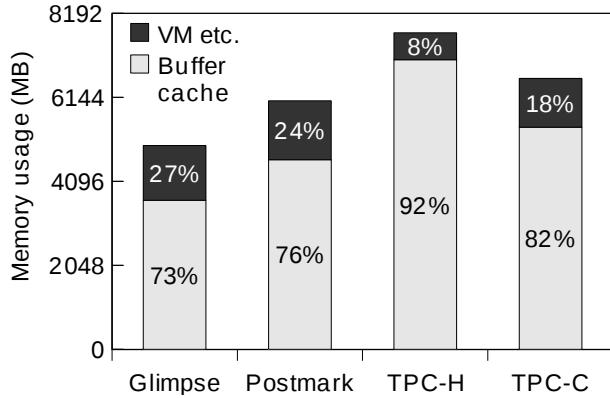
## 1 Introduction

Modern computer systems contain many energy hungry components; however the energy consumption of main memory is surpassing other components. Energy consumption of main memory mainly depends on its capacity and bus frequency. Bus frequencies are steadily rising to provide higher data bandwidth to the data hungry multi-core processors. In addition, demand for memory capacity is growing at an even faster rate to accommodate the data-centric applications that dominate in today's data centers. For example, the recent EMC Symmetrix DMX3000 storage system can be configured to have up to 256 GB of main memory [4] and the IBM eSeries p5 595 server is configured with 2TB of main memory [12]. Subsequently, a significant portion of total energy is consumed by the main memory. For example, as much as 40% of the system energy is consumed by the memory system in a mid-range IBM eServer machine [11]. The demand for higher memory capacity is not limited to data servers. Even portable

computers are experiencing rapid growth in memory capacity to accommodate user demand for richer multimedia experiences and support for processing capability enabled by high performance processors and video cards.

Energy optimization of the memory subsystem is addressed at both the hardware and software level. On the hardware level, energy efficiency is primarily gained through advances in manufacturing processes resulting in the creation of denser modules and a lower per-bit energy consumption. Hardware designers also added low-power modes to the modern SDRAM and exposed them to the system software, enabling OS-driven power management. The operating system has a detailed view of the system demand and therefore allows more sophisticated power management mechanisms to be implemented. However, the task is not easy since transitions between those power modes incur high overheads in both time and energy. Performance or energy efficiency may suffer if the transitions are not handled properly.

In modern operating systems, main memory is dynamically distributed between the buffer cache and the virtual memory available to the executing applications. The buffer cache is critical, since it can occupy as much as 77% of total available memory on desktop computers and much more on storage servers [10]. A majority of file-I/O accesses to the buffer cache are made through system calls and therefore the operating system knows when the access completes and can put the memory into a power saving state immediately. While this simple method saves energy, the transition delays upon the next access can significantly degrade memory performance. Subsequently, we focus on eliminating the delays incurred by power state transitions while maintaining high energy savings for the buffer cache. Specifically, our contributions are as follows: (1) we quantify the need for advanced techniques for reducing overheads of memory power state transitions in buffer cache, (2) we are the first to design accurate mechanisms for hiding delay in buffer cache power management for all access types, (3) we quantify the benefits and overheads of each proposed mechanism and heuristic through detailed energy and delay simulation study.

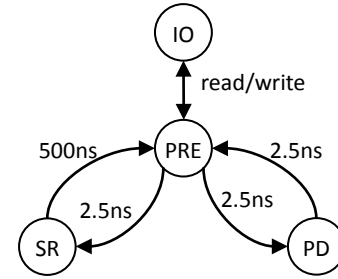


**Figure 1. Memory profile for I/O bound workloads running in a 8GB-RAM system.**

## 2 Background

Physical memory is allocated between the virtual memory (VM) and the buffer cache in a computer system. The buffer cache improves the file system performance by caching the previously accessed disk blocks. As data intensive applications become prevalent, the demand placed on the file system increases. As a result, buffer cache in modern data servers can span terabytes of physical memory [12]. Figure 1 shows a profile of memory usage during the execution of several I/O bound workloads in a Linux-x64 system with 8 GB RAM. As we can see, as much as 92% of memory usage is allocated for the buffer cache while less than 27% is used for virtual memory and other system memory. It is evident that the buffer cache accounts for a large amount of memory use in I/O bound applications. Subsequently, power management for the buffer cache is crucial to improving the overall memory energy efficiency. Furthermore, main memory is a central system component, and any performance degradation can have a significant impact on overall system performance. Therefore, improvements in energy efficiency should avoid introducing any delays into memory operations.

The simple approach to preventing delays is to keep all memory currently owned by a given process in a high-power state. In the case of virtual memory [6], only memory devices used by the newly-scheduled process are powered up during the context switch and remain on during the execution of the given process to prevent any performance degradation. Per-process power management has been further extended to the buffer cache [10], where memory devices that contain the buffers used by the running process is powered on and kept on during the process execution. While per-process approaches provide significant benefits in multitasking environment, they do not address memory management issues of server environments, where usually only one server process is running. In this scenario, all memory devices that belong to this single process would



**Figure 2. Power state transitions and latencies for Micron 1Gbit DDR2-800 devices.**

always remain on, eliminating any possible energy savings.

Unlike virtual memory, every access to the buffer cache through system calls is visible to the OS, allowing for more sophisticated mechanisms that can take advantage of this additional context. By separating the VM power management from the buffer cache power management, we can achieve better energy efficiency and performance for buffer cache through leveraging I/O system calls. In this paper, we only consider standard buffer cache accesses made through systems calls; any file that is memory mapped to the VM space is managed by the VM power management, and is not impacted by the buffer cache power management. Furthermore, separation of the buffer cache and virtual memory into separate spaces in physical memory can also improve overall energy efficiency [10]. The mechanisms proposed in this paper likewise allocate the buffer cache in a separate portion of physical memory to prevent interference from VM accesses.

SDRAM is widely used for main memory in the form of Double-Data-Rate (DDR), followed by DDR2 and DDR3. We focus on the DDR2 in this paper since it is the mainstream DRAM architecture in today's computer systems. DDR2 is packaged into DRAM modules, each of which commonly consists of two ranks. Each rank includes a number of physical devices and, in the case of server systems where timing and data integrity are a critical consideration, the registers and phase-lock loop (PLL) devices. The smallest unit of power management in DDR2 is the rank and all devices in a rank are operating at the same power state [13]. For simplicity, we only consider four power states that a rank can operate in: (1) I/O state: the state when memory is reading or writing data; (2) Precharge state (PRE): the active idle state where the next I/O can take place immediately at the next clock; (3) Powerdown state (PD): several subcomponents of a rank are disabled to save power, such as sense amplifier, row/column decoder, etc; (4) Self Refresh state (SR): in addition to the Powerdown state, the PLL and registers are disabled, further reducing power consumption. Since memory I/O can only be performed in the active state (PRE), a rank in a low-

States	Power [mW]		
	Device	PLL & REG	Rank
Read	496	867.5	4835.5
Write	644	867.5	6019.5
Precharge	126.2	867.5	1877.1
Powerdown	12	867.5	963.5
Self Refresh	10	0.075	80.075

**Table 1. Power specifications for a rank consisting of 8 Micron 1Gbit DDR2-800 devices.**

power state (PD or SR) has to transition to PRE state before performing any I/O, incurring an associated resynchronization cost. Figure 2 illustrates the latencies corresponding to the power state transitions for a DDR2-800 rank. Table 1 presents the corresponding power states [14, 15] with the total rank power being calculated as the product of the per-device power and the number of devices in the rank, plus the power consumed by the registers and PLL.

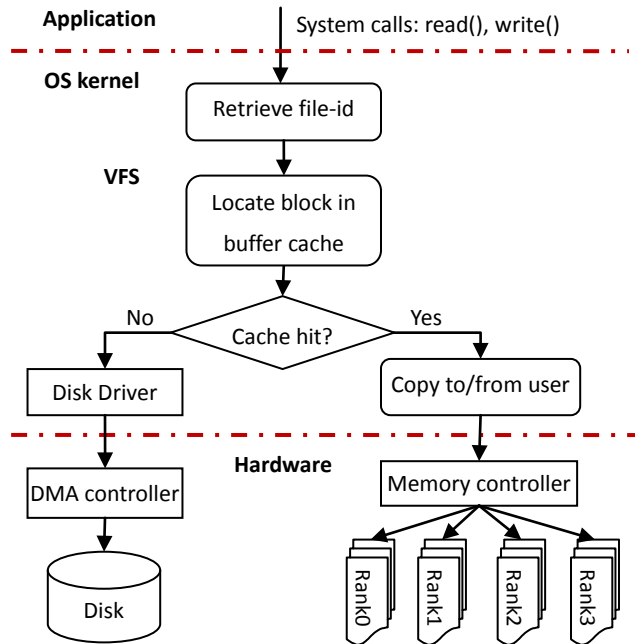
Physical memory accesses are bursty, and thus the majority of energy is consumed during the long idle time. Energy consumption can be significantly reduced by transitioning ranks to a low-power state during idle periods. Immediate Self Refresh (ISR) [8] is an on-demand mechanism that powers on a rank upon a physical memory request, and powers it off after the request completes. ISR can significantly degrade memory performance since every memory access incurs the full resynchronization latency. For example, reading a 4 KB buffer block in DDR2-800 memory (6.4 GB/s bandwidth) in SR state would degrade the performance by 82%, due to the 500 ns transition latency to PRE state. Therefore, it is critical for power management mechanisms to prevent the delays associated with accessing ranks in low-power states.

### 3 Design

Hardware-level power management may benefit from fine-grained, low-level information provided by the memory controller that monitors the memory activities on each device and turns the devices off based on the detection of idle periods. However, hardware is generally unaware of the high-level information, such as which file is accessed by which process or the layout of file blocks in the buffer cache. As a result, hardware-level power management may suffer from inaccuracy and can expose the application to unexpected performance degradation. System-level power management, on the other hand, can provide the necessary context to make timely power-state transitions and reduce or eliminate performance losses.

#### 3.1 Exploiting File-I/O System Calls

A file-I/O system call goes through several steps in the kernel before the actual buffer cache data is read from phys-



**Figure 3. Anatomy of File-I/O operations.**

ical memory, as shown in Figure 3. Upon the kernel entry, the file table is checked to get the file identifier (inode), and the inode is used to access the virtual file system (VFS). Each requested block of the file I/O is looked up in the hash table that tracks all blocks in the buffer cache. If the block is found in the cache, the kernel invokes `_copy_to_user` routine to copy the block from the kernel space to the destination location in user space (`_copy_from_user` is called for write operations), and the corresponding I/O request is sent to the memory controller.

Energy consumption can be reduced by putting the rank into a low-power state during the idle period between two consecutive I/Os. To avoid powering down rank that will be immediately accessed, power management techniques for VM use a timeout scheme that usually waits for a few nanoseconds for more memory I/Os to arrive before powering down the rank [5]. The access granularity for file I/O is coarser: the idle period between two consecutive memory I/Os in buffer cache is on the order of hundreds of microseconds. Therefore to increase energy savings further, the mechanisms proposed in this paper immediately power down the accessed ranks as soon as a file-I/O finishes.

Due to the high resynchronization cost, it is critical to determine when to turn on a rank upon the access request to the rank. The best scenario occurs when the transition finishes just before the request arrives at the memory controller. In this situation, no energy is unnecessarily consumed while waiting for the request and the transition latency is completely hidden. Late transitions delay memory I/Os and transition latencies are partially or fully exposed.

The overall system energy consumption can also increase due to the longer service time. Finally, if the rank is turned on earlier than request arrival, the delay is avoided at the cost of excess energy consumed when waiting for the request. As shown in Figure 3, there are several stages where the rank can be turned on. The goal is to initiate and complete the transition before a memory request arrives at the memory controller.

We have instrumented the Linux 2.6.20 kernel and measured the time spent within each I/O call subroutine on a 3.0GHz AMD processor. We first investigated the time between block-locating routine and `_copy_to_user` routine, where 90% of the system calls complete in fewer than 120 ns, far from hiding the 500 ns transition latency from Self Refresh state. Transitions initiated at this time will be too late, exposing almost 76% of transition delays. We then investigated the time between file-id retrieving routine and `_copy_to_user` routine, finding that for 90% of I/Os the time is longer than 500 ns, and therefore, can fully cover the transition latency. Furthermore, 85% of I/Os encounter times between 500 ns and 550 ns. These key observations strongly suggest that turning on ranks right after file-id retrieving routine can preserve 90% of performance with hardly any energy penalty. Since file-id retrieving routine only takes a few instructions in the beginning of a file-I/O system call, we will refer to this point as "system call entry" for the remainder of the discussion.

### 3.2 All-Ranks-On for Delay Reduction

Physical memory mapping is not available at the system call entry, and performing a fully fledged translation is not feasible at this point. Subsequently, we propose a simple ALL-Ranks-ON mechanism (ALL) that will naively turn on all ranks containing buffer cache data at the system call entry. The ALL mechanism provides high performance for memory accesses since all the ranks are ready before a memory request arrives and can serve the request without delays. However, file blocks requested by a I/O generally reside on a single rank. Other ranks that do not hold the data are unnecessarily turned on and consume energy during the I/O operation. While energy savings are obtained by keeping memory in the low-power state during the idle periods between I/Os, these periods can be small in data intensive applications. Therefore, the ALL mechanism may offer less than desired energy savings. Nevertheless, the ALL mechanism represents a lower bound on the exposed transition delay for all the proposed mechanisms that power on the rank at the system call entry.

### 3.3 Predicting Individual Ranks

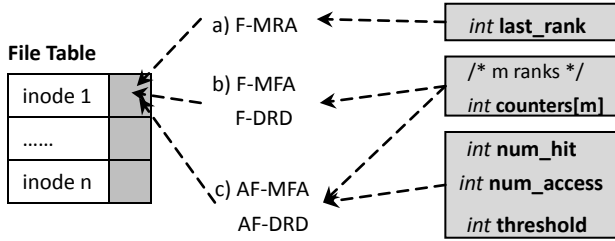
To reduce the energy consumption of the ALL mechanism, we will propose several mechanisms that predict the

most likely rank to be accessed and only turn on that rank at the system call entry. The simplest mechanism is the Most-Recently-Accessed (MRA) mechanism that predicts the current I/O will access the same rank as the last I/O. With sequential cache allocation, buffers are allocated in the order they are accessed, and an entire rank is filled before moving on to the next one. In case of sequential access applications that generally read an entire file sequentially, the MRA mechanisms will achieve high accuracy since it is highly probable that the current referenced blocks are allocated to the same rank as the previous one.

The MRA mechanism can potentially save more energy since only one single rank is predicted and turned on for each I/O. However, in random access applications, its accuracy could be drastically reduced, which increase both delays and energy consumption. When an incorrect rank is turned on, the transition of the actual accessed rank is delayed until the request arrives at the memory controller, exposing the full transition delay. To refine the prediction granularity, and hopefully increase the accuracy, we can distribute the last rank information to each individual file. Subsequently, we propose the per-File MRA (F-MRA) mechanism that predicts and turns on the last used rank by each accessed file. To implement F-MRA, we slightly modify the file table in the kernel to include the information of last used rank for each file (shown in Figure 4a.)

The F-MRA mechanism will benefit applications that access many files concurrently but accesses within each file are sequential or at least grouped in the same rank. However, problems may arise when the accessed file is spread among several ranks due to replacements or access patterns that did not result in contiguous placement in the rank. Therefore, we consider probabilistic approaches and propose the per-File-Most-Frequently-Accessed (F-MFA) predictor that keeps track of the per-file access frequencies. To count the rank access frequency for each file, we add an array of counters in the associated file table entry (shown in Figure 4b.). Each of the counters corresponds to one rank and records the number of completed accesses in that rank. The counter value is incremented upon an access to the corresponding rank. Each time a file is accessed, the F-MFA mechanism predicts that the file's most frequently used rank (with the largest counter value) will be accessed.

Frequency based approaches suffer sometimes from hot blocks, where a block is initially accessed very frequently and not accessed in the future. As a result, we propose another mechanism based on file distribution among ranks. The per-File-Data-Rank-Density (F-DRD) predictor keeps track of the number of blocks that each rank contains for a given file. The data structures in each file table entry are identical to the F-MFA mechanism, as shown in Figure 4b. The only difference in the implementation is that the counters are updated upon each buffer placement and replace-



**Figure 4. Data structure (shaded area) added to the file table for proposed mechanisms.**

ment in the given rank for F-DRD. Every time a new block is allocated to a rank, the corresponding counter is incremented by 1. When a block is replaced out of cache, the counter corresponding to its original rank is decremented by 1. Similarly to the F-MFA mechanism, the F-DRD mechanism retrieves the file distribution from the file table and turns on the rank that contains the most blocks of the currently accessed file.

### 3.4 Increasing Prediction Coverage

Both F-MFA and F-DRD mechanisms predict only one rank upon a system call. As a result, they can miss turning on the accessed ranks and expose the delays. To reduce the misprediction, we increase the number of ranks that each mechanism turns on. The simplest extension to those mechanisms is setting a desired threshold  $T$ . For F-MFA,  $T$  represents the minimum access frequency that all powered-on (active) ranks should provide. For F-DRD,  $T$  represents a minimum file fraction to be present in active ranks. For each I/O, F-MFA will select to turn on the smallest number of ranks that can provide the total access frequency greater than  $T$ . For example, if we set the threshold  $T$  to 90% and the file is distributed among 3 ranks with access frequencies of 62%, 30% and 8%, F-MFA will turn on the first two ranks, providing the minimum desired frequency of 90%. Similarly in case of F-DRD, it will turn on the minimum number of ranks that provide the total file fraction greater than  $T$ . For example, if we want 90% of the file to be present in active ranks and the file is distributed among 3 ranks with the block distribution of 62%, 30% and 8%, F-DRD will turn on the first two ranks containing a total 92% of the file blocks.

Setting the threshold to 100% will turn on all ranks that the accessed file resides in, eliminating all delays while keeping ranks that do not contain any portion of the file in low-power states. However, some applications may be able to better tolerate the delay and 100% of accesses to active ranks may not be necessary. For example, the user may desire a global coverage rate of 90% by the predictor, meaning that 90% of the accesses should go to active ranks that are powered on by the predictor. To enable the user desired global coverage rate, we modify F-MFA and F-

DRD and propose Adaptive F-MFA (AF-MFA) and Adaptive F-DRD (AF-DRD) that adaptively adjust the threshold for each file, i.e., the minimum required access frequency or block distribution that active ranks should provide. Subsequently as shown in Figure 4c, these mechanisms maintain three additional variables per file in the file table: 1) the number of accesses that resulted in hits in active ranks ( $num\_hit$ ), 2) the number of completed accesses to the file ( $num\_access$ ), and 3) the current threshold ( $threshold$ ). Initially, the user presets the desired global coverage rate, and each file's  $threshold$  is set to some default value, e.g. 50%. After every access,  $threshold$  is dynamically adjusted based on the comparison of the local coverage rate (calculated as  $num\_hit$  divided by  $num\_access$ ) to the global coverage rate. If local coverage rate is lower,  $threshold$  is increased and otherwise decreased. Overtime, the local coverage rate will eventually converge to the global coverage rate.

A similar optimization can be implemented for the F-MRA mechanism, resulting in the Adaptive F-MRA (AF-MRA) mechanism. In AF-MRA, we also introduce three additional variables in each file table entry, as well as a stack to maintain the list of MRU ranks. The first two variables maintain the number of accesses and the number of hits as before to calculate the local coverage rate. The third variable  $num\_ranks$  stores the number of most recently used ranks that the predictor should turn on. Initially  $num\_ranks$  is set to one, and it is automatically adjusted based on the comparison between local coverage rate and the global coverage rate: if the local coverage rate is lower,  $num\_ranks$  is increased, and otherwise decreased. Upon each file-IO, AF-MRA will turn on the most recently used ranks indicated by  $num\_ranks$  and the MRU stack.

### 3.5 Improving Energy Efficiency

The proposed mechanisms may turn on ranks that will not be accessed, either by mispredicting a rank to turn on, or by aggressively turning on more than one rank. Subsequently, the wrongly turned-on ranks will be idling in Precharge state during the I/O, and consume excess energy. To address this issue, we propose an early-turnoff optimization that can track the rank to be accessed and turn off the unneeded active ranks at the earliest point to save energy. To efficiently select the rank to be accessed from active ranks, we slightly modify the data structures in the buffer cache hash table. For each table entry, we add an additional variable to record the rank that the corresponding block is placed in. Subsequently, when a block is requested, its containing rank is to be accessed and can be obtained as soon as the buffer cache lookup completes in block-locating routine. Other active ranks will not be accessed and can now be turned off. With this optimization, each proposed mechanism executes the following steps. First, at the system call entry, several ranks are turned on as indicated by the pre-

Application	Number of I/O system calls	Number of referenced blocks	Number of unique files	Dataset size (MB)	Hit ratio (4GB)
Glimpse	18351930	20574810	319712	4237	84%
Postmark	17897551	20000000	101421	5047	94%
TPC-H	98789470	100000000	107	7156	96%
TPC-C	47461324	95908075	542	4775	97%
Multi1 (TPC-H, TPC-C)	197410317	200000000	112	7247	96%
Multi2 (Glimpse, Postmark)	26867682	30000000	366221	8500	86%

**Table 2. Applications and trace statistics.**

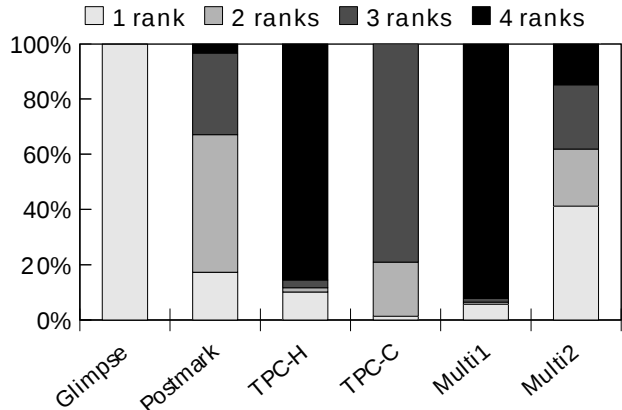
dicator. Second, once the buffer cache lookup is performed, the rank to be accessed is retrieved and all the other ranks that were previously turned on are turned off immediately.

## 4 Methodology

We used trace-driven simulation to evaluate our proposed mechanisms and compared them with the existing on-demand mechanism (ISR). Detailed file-I/O traces of each application were collected by a modified *strace* utility that intercepts the system calls from the traced process and records the following information about each I/O operation: access type, time, file identifier, and I/O size. We selected four common server applications: Glimpse, Postmark, TPC-H, and TPC-C. Glimpse is a text indexing and search application. Postmark is a mail server benchmark which is designed to measure the transaction rates for a workload approximating a large mail server. TPC-H is a decision support benchmark which consists of a suite of business oriented ad-hoc queries and concurrent data modifications. TPC-C is an on-line transaction processing benchmark, simulating a intense computing environment where numerous users execute transactions against a database. We also generated two concurrent traces Multi1 and Multi2. Multi1 is a trace of TPC-H and TPC-C executed concurrently and represents a workload in a database querying and processing system. Multi2 is a trace of Glimpse and Postmark executed concurrently and represents a web server providing mail and web search.

Table 2 lists the details of the each trace, such as the number of file-I/Os, the number of referenced blocks, the number of files accessed, dataset size, and the hit rate in a 4 GB buffer cache. Due to the long runtime of Postmark and TPC-H, we selected the first 20 million references for Postmark and the first 100 million references for TPC-H in our traces. The concurrent traces were also truncated accordingly; as a result the statistics of concurrent execution is not the sum of the two individual applications.

We augmented a buffer cache simulator [1] with a physical memory simulator to implement the proposed mechanisms. The memory simulator includes a memory controller and 4 DDR2-800 ranks to simulate a total 4 GB



**Figure 5. Access frequency of all files based on the file distribution over ranks.**

memory with the power specification shown in Table 1. The entire storage space is used for buffer cache with 4KB block size. Since the traces do not contain the I/O processing time in the kernel, we set the time from the system call entry to `_copy_to_user` routine to 500 ns, and set the time from block-locating routine to `_copy_to_user` routine to 120 ns, as discussed in Section 3.1. Finally, sequential buffer placement is used to allocate free buffers and the LRU replacement algorithm is used to select the victim block during the buffer replacement.

## 5 Evaluation

### 5.1 Accuracy of Predictors

Figure 6 shows the prediction breakdown for the proposed mechanisms. The Hit portion represents the number of transitions of ranks that were correctly turned on by the predictor, while the Miss portion represents the number of transitions of ranks that were accessed but not turned on by the predictor. The sum of hits and misses is the total required rank transitions in each application, and is normalized to 1. The Wrong portion above 1 represents the number of rank transitions that were initiated by the predictor but not accessed in the subsequent file-I/Os. Therefore, a missed rank transition results in exposed transition delay, while a wrong rank transition consumes excess energy.

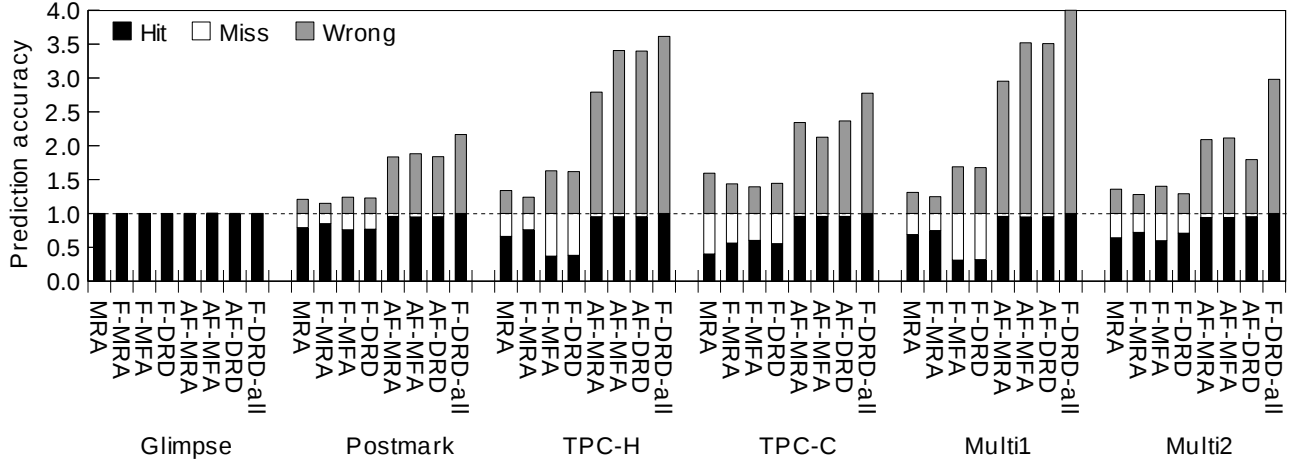


Figure 6. Prediction accuracy of the proposed mechanisms.

Lets first focus on the predictors (MRA, F-MRA, F-MFA, F-DRD) that only predict to turn on one rank at a file-I/O. Each of these single-rank predictors has the same number of wrong transitions as the number of misses, meaning that a predictor with more hits will have less wrong transitions. For Glimpse, all file blocks are sequentially read and allocated, and as a result, all predictors are almost 100% accurate in predicting ranks with hardly any wrong transitions. For other random access applications, F-MRA achieves the highest accuracy, on average 73% hits with only 27% wrong transitions. MRA also predicts to use last rank, but due to the aliasing among different files, the accuracy drops to 64%. The mediocre accuracy of frequency (F-MFA) and file-distribution (F-DRD) based mechanisms show that temporal information provided in MRA and F-MRA is more important for achieving higher accuracy.

When the files are distributed across multiple ranks, it becomes more difficult to accurately select a single rank. Figure 5 shows access frequency to files distributed over multiple ranks. The majority of files in Glimpse are small and, due to sequential accesses, each of them ends up in a single rank. Therefore, I/O accesses in Glimpse always go to files distributed in one rank, explaining the excellent performance of all the predictors. Files in Postmark are also small, ranging from 50 Bytes to 100 KB, but random accesses prevent contiguous allocation of files to ranks. As a result, files are scattered in multiple ranks and 80% of the I/Os go to the files distributed in two or more ranks. Multi2 is a mix of Glimpse and Postmark and shows the intermediate pattern. In TPC-H, TPC-C and Multi1, large database files are accessed randomly such that the blocks of a large file are allocated across multiple ranks. Therefore, 80%-90% of the I/Os from these three applications go to the files distributed in three or more ranks.

Coverage rate in random-access applications can be increased by turning on more ranks at the system call entry, increasing the likelihood that the file portion accessed by

current I/O resides in active ranks. Through experiments of F-MFA's file-access-frequency thresholds and F-DRD's file-fraction thresholds valuing between 50% and 100%, we found that the resulting coverage rate is 95% on average for both predictors. Subsequently, we use 95% as the global coverage rate for the study of the adaptive predictors. Figure 6 shows that all adaptive predictors (AF-MRA, AF-MFA and AF-DRD) match the preset coverage rate of 95%, significantly improving the coverage of the single-rank predictors. However, higher coverage comes at the cost of more unnecessary transitions, and as a result, wrong transitions in AF-MRA, AF-MFA and AF-DRD increase by 3.2, 1.4, and 1.5 times as compared to F-MRA, F-MFA, and F-DRD, respectively. AF-MRA is the best adaptive predictor due to its smallest wrong transitions. Finally, we set AF-DRD's global coverage to 100%, and shows the effects of the resulting F-DRD-all mechanism. By turning on all ranks that contain any part of the file, F-DRD-all yields the maximum 100% coverage at the expense of the highest wrong transitions among all predictors.

## 5.2 File-I/O Time

Both performance and energy efficiency of the entire system is affected by performance degradation of main memory. Therefore, the goal of the proposed mechanisms is to improve the performance while preserving energy savings. The performance of file-I/Os is shown in Figure 7. The overall I/O time for each mechanism and application is separated into three components: 1) memory I/O time: time spent on reading/writing data, 2) kernel time: time spent on processing file-I/O system calls inside the kernel, and 3) delay: additional time spent in transitioning ranks from Precharge state to Self Refresh state before any I/O can be performed in physical memory. Since we focus on improving the system call turnaround time, i.e., the time between the system call entry and the return to the application, we exclude the idle time between consecutive file-I/Os when

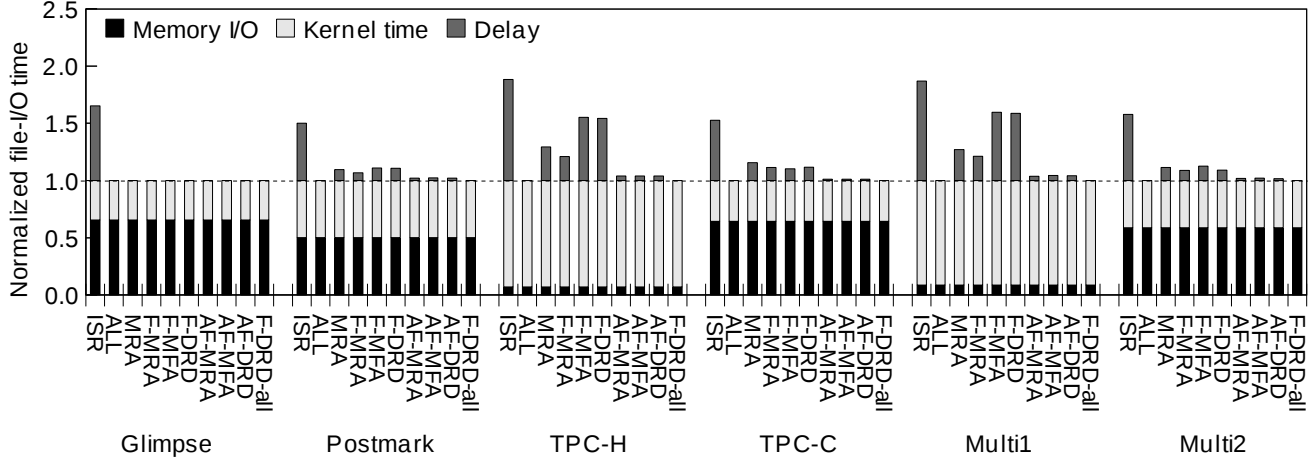


Figure 7. Breakdown of file-I/O time.

evaluating the performance. Finally, neither the memory access time nor the kernel time are affected, since we do not alter the processing routines.

Ideally, file-I/O time only contains the kernel time and memory I/O time. Therefore, we normalize each mechanism’s time to this ideal time and the delay is shown above the 1.0 line. As we can see, ISR has the most delay since every rank transition is delayed until physical memory access, and as a result incurs 50%-87% performance degradation across all applications. On the other extreme, ALL incurs zero delay by turning on all ranks in the system at the system call entry. The other mechanisms complete the spectrum of tradeoffs between energy and delay. Higher coverage of a predictor (the Hit portion of Figure 6) translates into less delay. This is because a rank that was missed turning on will cause an on-demand transition and incur the full resynchronization latency, while the correct prediction will complete the rank transition before the memory request arrival and fully hide the delay.

For Glimpse, all predictors have almost 100% coverage, thus eliminating all delay. In other random access applications, lower coverage of single-rank predictors results in higher delays. Subsequently, MRA, F-MRA, F-MFA, and F-DRD incurs performance degradation of 19%, 14%, 30%, and 29% respectively. Since all of the adaptive predictors (AF-MRA, AF-MFA, and AF-DRD) achieve 95% coverage rate, they significantly reduce delays, resulting in a mere 3% performance degradation. Finally, F-DRD-all turns on all ranks containing any portion of the file, and therefore preserves the full performance.

### 5.3 File-I/O Energy Consumption

Figure 8 shows the energy consumption of physical memory during the processing of file-I/Os from the system call entry until exit. The energy bars are divided into four components: 1) energy consumed for reading/writing data, 2) energy consumed in Precharge state, 3) energy con-

sumed in Self Refresh state, and 4) energy consumed during the transitions between Precharge state and Self Refresh state. We normalize each mechanism’s energy consumption to ISR’s energy (shown as 1.0), since the on-demand ISR is the most energy-oriented power management mechanism when we only consider the energy consumption of main memory. It is worth noting that delays introduced by ISR may increase the overall energy consumption of the entire system. If we considered overall system energy, the lower delays incurred by the proposed mechanisms would result in better overall energy efficiency.

Energy consumed by memory I/O is the same across all mechanisms for any given application, since the buffer cache reads/writes the same amount of file data. The energy consumed in other states, however, varies among different mechanisms. The largest differences are visible in Precharge energy and Self Refresh energy. ALL consumes the most amount of Precharge and transition energy since it turns on all ranks at each file-I/O and the ranks remain in Precharge state during the entire I/O. The energy overheads of the other predictors are proportional to their prediction coverage as shown in Figure 6. Single-rank predictors: MRA, F-MRA, F-MFA, and F-DRD, each reduce ALL’s Precharge energy by 46%, 57%, 11% and 10%, respectively, with only a small increase in Self Refresh energy. Finally, they respectively yield a 33%, 37%, 21% and 21% reduction in total energy consumption, as compared to ALL. However, energy efficiency is achieved at a cost of higher delays as shown in Figure 7. Adaptive predictors AF-MRA, AF-MFA, and AF-DRD reduce delays more aggressively by turning more ranks on and consume on average 64%, 33%, 29% more Precharge energy than the single-rank predictors.

An interesting scenario occurs in TPC-H and Multi1 where the single-rank predictors F-MFA and F-DRD consume more Precharge energy than the other mechanisms. The explanation is twofold. First, the 37% coverage for

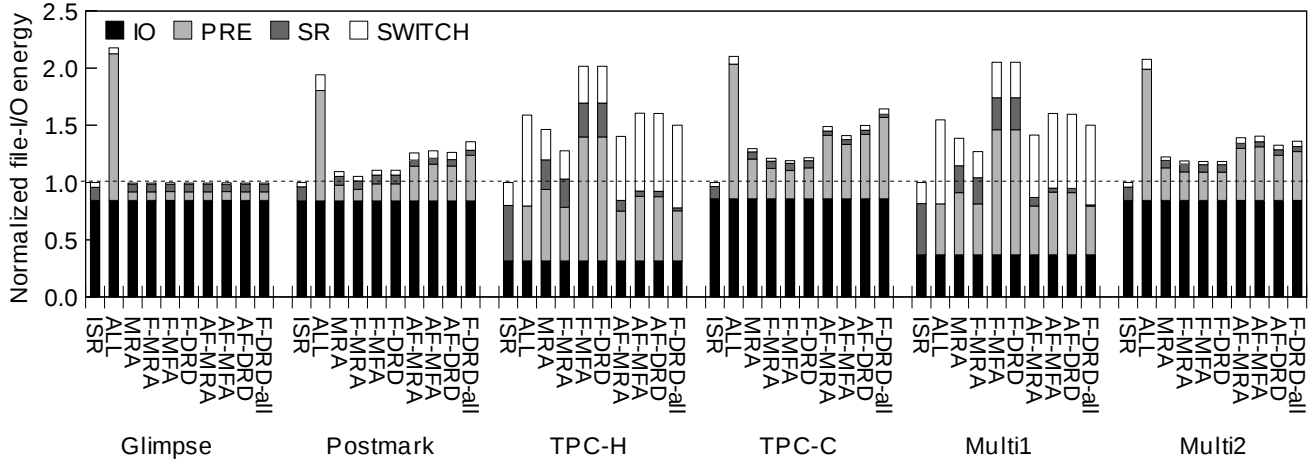


Figure 8. Breakdown of memory energy consumption.

F-MFA and F-DRD is the lowest, incurring the most on-demand transitions delayed until the physical memory accesses. Second, the average I/O size in TPC-H is around 66 bytes which only takes 9.8 ns for DDR2-800, and as a result, the 500 ns transition delay incurred by a misprediction prolongs the I/O time by 51 times. Therefore, during the severely prolonged I/O period, wrongly transitioned ranks consume higher Precharge energy and other ranks consume more Self Refresh energy, resulting in much higher total energy consumption.

The cases of TPC-H and Multi1 illustrate the negative impact of delays not only on performance but also on energy savings. We can expect similar behavior from other power management mechanisms that sacrifice performance for energy savings of a single component, but may cause the entire system to consume more energy. Therefore, it is critical for power management mechanisms to focus on reducing delays to minimize the impact of performance degradation on the overall energy efficiency.

#### 5.4 Energy Reduction with Early-Turnoff

The energy consumed by wrongly transitioned ranks can be reduced by turning them off earlier, as described in Section 3.5. Figure 9 compares the effectiveness of this optimization for adaptive predictors. Each mechanism with early-turnoff has a suffix et, and its energy consumption is still normalized to ISR. Significant benefit is achieved from early-turnoff optimization that eliminates Precharge energy consumed by unused ranks. Subsequently, the early-turnoff technique reduces the total energy consumption by 27%, on average, for all mechanisms shown in Figure 9. The best mechanism with early-turnoff, AF-MRA-et, only consumes 3% more energy than ISR when only considering memory energy. The energy efficiency would be actually better than ISR if the entire system energy was considered.

## 6 Related Work

Power management techniques for main memory can be generally grouped into hardware approaches and software approaches. Among hardware approaches, Lebeck et al. [9] studied the interaction of page placement with static and dynamic hardware policies to reduce the memory power dissipation by using extensive simulation. The cooperation between the hardware and the OS was also studied in [9]. Furthermore, Pisharah et al. [17] proposed another approach to save memory energy by introducing a hardware called Energy-Saver Buffers to mask the resynchronization costs when reactivating memory modules. Pandey et al. [16] showed that significant energy is consumed when memory is actively idle during DMA transfers and proposed mechanisms that increase the level of concurrency between multiple DMAs. Based on this work, Yue et al. [18] targeted the buffer cache in data servers, and evaluated the energy efficiency for various buffer cache replacement algorithms under real-world parallel I/O workloads through simulation.

Delaluz et al. [2] proposed a compiler-directed approach to cluster the data across memory banks, detect memory module idleness and insert power-state transition instructions into a program by offline profiling. Delaluz et al. [3] also proposed an operating system based solution where the OS scheduler directs the power mode transitions by keeping track of module accesses for each process in the system. Subsequently, Huang et al. [6] proposed Power-Aware Virtual Memory that manages power of memory devices on per-process basis. Based on PAVM, a SW-HW cooperated mechanism citepmu was proposed to combines PAVM and the underlying hardware. Huang et al. [7] also proposed memory reshaping mechanisms that coalesce short idle periods into longer ones through page migration. Finally, Li et al. [12] proposed a mechanism that can guarantee the performance by temporarily stopping power management and running memory at full performance.

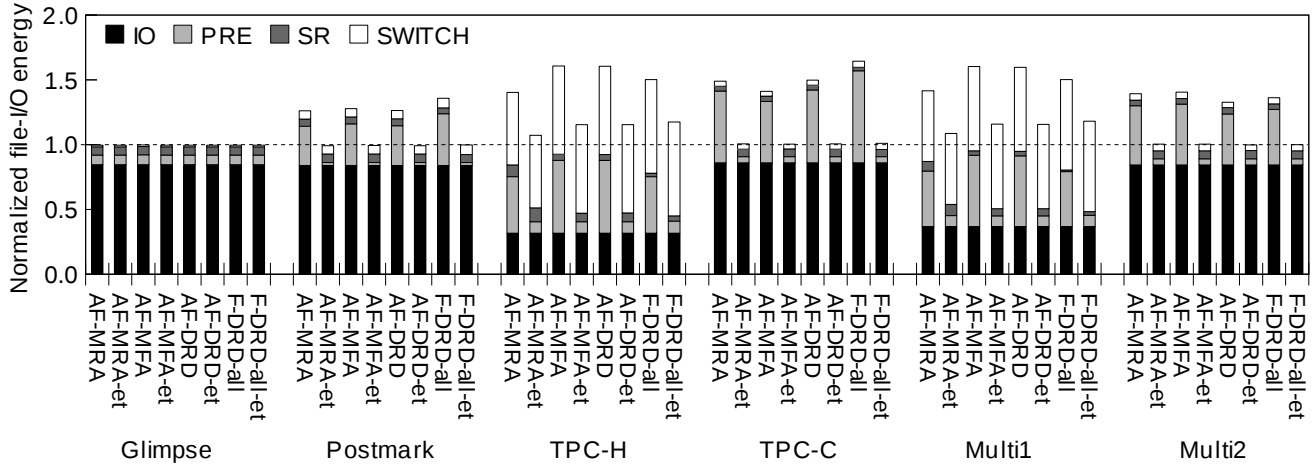


Figure 9. Energy consumption before and after early-turnoff.

## 7 Conclusions

Due to the increasing demand for large-scale data processing, memory has become one of the most energy-hungry components in the computer system. Data intensive applications require an efficient I/O subsystem and as a result the demand for a larger buffer cache is growing, surpassing the storage demand of virtual memory. Therefore, a low-power and high performance buffer cache is important for the overall system efficiency. To improve performance of power management mechanisms while providing high-energy savings, we presented several delay-hiding mechanisms for power state transitions of DRAM. We have shown that power state transitions can be efficiently hidden by standard I/O processing routines, and leverage that fact in proposing a range of mechanisms that efficiently hide transition delays. The best-proposed mechanism with early-turnoff energy optimization can hide almost all transition delays while only consuming 3% more energy than existing mechanisms.

## 8 Acknowledgments

This research was funded by the National Science Foundation under Grant No. 0844569.

## References

- [1] A. R. Butt, C. Gniady, and Y. C. Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *SIGMETRICS*, 2005.
- [2] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Hardware and software techniques for controlling dram power modes. *IEEE Transactions on Computers*, 2001.
- [3] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Scheduler-based dram energy management. In *DAC*, 2002.
- [4] EMC. Emc symmetrix dmx 3000 specification sheet., April 2005. [http://www.emc.com/products/systems/symmetrix/symmetri\\_DMX1000/pdf/DMX3000.pdf](http://www.emc.com/products/systems/symmetrix/symmetri_DMX1000/pdf/DMX3000.pdf).
- [5] X. Fan, C. Ellis, and A. Lebeck. Memory controller policies for dram power management. In *ISLPEd*, 2001.
- [6] H. Huang, P. Pillai, and K. G. Shin. Design and implementation of power-aware virtual memory. In *ATEC*, 2003.
- [7] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller. Improving energy efficiency by making dram less randomly accessed. In *ISLPEd*, 2005.
- [8] H. Huang, K. G. Shin, C. Lefurgy, K. Rajamani, T. W. Keller, E. V. Hensbergen, and F. L. R. III. Software-hardware cooperative power management for main memory. In *PACS*. Springer, 2004.
- [9] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. *SIGPLAN Not.*, 2000.
- [10] M. Lee, E. Seo, J. Lee, and J.-s. Kim. Pabc: Power-aware buffer cache management for low power consumption. *IEEE Transactions on Computers*, 2007.
- [11] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *Computer*, 2003.
- [12] X. Li, Z. Li, Y. Zhou, and S. Adve. Performance directed energy management for main memory and disks. *Transactions on Storage*, 2005.
- [13] Micron. <http://download.micron.com/pdf/datasheets/dram/ddr2/1GbDDR2.pdf>.
- [14] Micron. <http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf>.
- [15] Micron. [http://download.micron.com/downloads/misc/ddr2\\_power\\_calc\\_web.xls](http://download.micron.com/downloads/misc/ddr2_power_calc_web.xls).
- [16] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini. Dma-aware memory energy management. In *HPCA*, 2006.
- [17] J. Pisharath and A. Choudhary. An integrated approach to reducing power dissipation in memory hierarchies. In *CASES*, 2002.
- [18] J. Yue, Y. Zhu, and Z. Cai. An energy-oriented evaluation of buffer cache algorithms using parallel i/o workloads. *IEEE Transactions on Parallel and Distributed Systems*, 2008.