IAMEM: Interaction-Aware Memory Energy Management

Mingsong Bi
Intel Corporation
mingsong.bi@intel.com

Srinivasan Chandrasekharan
University of Arizona
schandra@cs.arizona.edu

Chris Gniady
University of Arizona
gniady@cs.arizona.edu

Abstract

Energy efficiency has become one of the most important factors in the development of computer systems. As applications become more data centric and put more pressure on the memory subsystem, managing energy consumption of main memory is becoming critical. Therefore, it is critical to take advantage of all memory idle times by placing memory in low power modes even during the active process execution. However, current solutions only offer energy optimizations on a per-process basis and are unable to take advantage of memory idle times when the process is executing. To allow accurate and fine-grained memory management during the process execution, we propose Interaction-Aware Memory Energy Management (IAMEM). IAMEM relies on accurate correlation of user-initiated tasks with the demand placed on the memory subsystem to accurately predict power state transitions for maximal energy savings while minimizing the impact on performance. Through detailed trace-driven simulation, we show that IAMEM reduces the memory energy consumption by as much as 16% as compared to the state-of-the-art approaches, while maintaining the user-perceivable performance comparable to the system without any energy optimizations.

1 Introduction

Modern computer systems ranging from netbooks to server clusters are relying on large system memory to provide high performance for data intensive applications. While a computer system contains many energy hungry components, the energy consumption of main memory is becoming more significant and can surpass energy consumption of other components. For example, as much as 40% of the total system energy is consumed by the memory subsystem in a mid-range IBM eServer machine [14]. The demand for higher memory capacity is not limited to data servers. Even portable computers are experiencing a rapid growth in memory capacity to accommodate user demand for higher processing capability and richer multimedia experiences. As a result, current portable systems, e.g. notebooks, are commonly sold with 8GB of main memory or more, and ultraportable systems such as netbooks with 4GB.

Energy optimization of the memory subsystem is being addressed at both hardware and software levels. At the hardware level, energy efficiency is primarily gained through advances in manufacturing processes to create denser modules and lower per-bit energy consumption. In addition, low-power states are also added to the modern SDRAM and are exposed to the system software, enabling OS-driven energy management. While energy management that utilizes multiple power states can be implemented in hardware, it is usually delegated to the operating system. The operating system has a detailed view of the running applications and the demand they place on the system, and therefore, allows for more sophisticated energy management. While the additional context available at the OS level provides better energy management possibilities, the task of designing an efficient energy management technique is not easy due to the long power-state transition delays that can be exposed to the application execution.

Performance and the overall energy efficiency may suffer if the overheads of power state transitions are not addressed properly, since the entire system has to stay on longer and consume additional energy. In addition, the user may even be irritated to the extent that he or she completely disables the energy management mechanisms. Fortunately, maximal memory performance is usually not necessary to meet the user's performance expectations. For example, CPU or I/O bound tasks in interactive applications may not be noticeably degraded when the memory is operating in a low-power state. Furthermore, the perceived performance of real-time applications such as video players, games, or teleconferencing may not be affected by the power state transitions, as long as the system maintains perceptual continuity for the user. Therefore, it is critical to distinguish the memory-intensive tasks that may expose transition delays to the users from the tasks with low memory activity. Subsequently, the former tasks must be executed with high memory performance, while the latter can be executed with lower performance to improve energy efficiency.

To take advantage of these opportunities, we propose Interaction-Aware Memory Energy Management (IAMEM), a highly accurate and transparent mechanism for memory energy management in interactive systems. Compared to the existing mechanisms: (1) IAMEM provides energy optimizations within the running process that the user is interacting with as well as all other processes in the system, as compared to previous approaches that only improved energy efficiency of memory occupied by processes waiting for execution [8, 13]; (2) IAMEM is a unified approach that addresses energy efficiency of both the buffer cache and the virtual memory, while previous approaches only proposed individual solutions for either the buffer cache or the virtual memory [2, 8, 13]. Subsequently, we make the following contributions in this paper: (1) identify and quantify the memory behavior of common interactive applications and show large opportunity for improvement; (2) utilize high-resolution context of user interactions to accurately predict memory demand for tasks initiated by the user; (3) propose IAMEM, a unified energy management mechanism for the entire memory subsystem; (4) compare IAMEM with the existing state-of-the-art mechanisms through a detailed study.

2 Motivation

Current trends in providing larger on chip CPU caches result in main memory seeing fewer accesses from the CPU, which creates longer memory idle times. Subsequently, the majority of memory energy is consumed in the idle state. Energy consumption of main memory can be significantly reduced by transitioning memory devices to a low-power state during the idle periods. However, accessing memory in a low-power state incurs high transition latency, and as a result, degrades the system performance and may increase the overall energy consumption. Therefore, it is crucial to ensure that the associated performance degradation can be hidden behind the application execution and not exposed to users.

A simple way to provide memory energy management is to keep the memory devices occupied by currently running process in a high-power state and power down all other memory devices. This per-process energy management is employed by Power-Aware Virtual Memory (PAVM) [8]. In this approach, the memory devices used by the newly scheduled process are powered up to provide high performance for the running process, during the context switch, while the other memory devices occupied by non-executing processes are kept in a low-power state to save energy. While this per-process

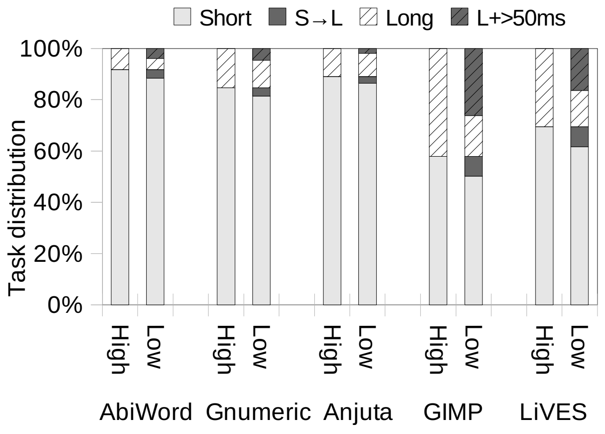


Figure 1: Distribution of tasks with memory in high and low power states. Shaded bar represents the fraction of short tasks extended over 50ms, while shaded-crossed bar represents the fraction of long tasks extended by more than 50ms.

approach provides significant benefits for a multitasking environment, it fails to address the energy consumption within a single process. Furthermore, current multicore CPUs with hyper-threading support can have tens of processes concurrently executing and accessing a wide range of memory addresses, rendering PAVM ineffective. To improve energy efficiency in such scenarios, we need a finer-granularity and more aggressive energy management that is able to reduce energy consumption within a single process during the process execution.

2.1 Per-Task Energy Management

Fortunately, the full performance is usually not needed in interactive applications, since users are unable to perceive certain amount of short delays. We can exploit that observation in designing more aggressive energy management mechanisms. Prior studies in human-computer interaction have established the human perception threshold to be between 50-100ms, indicating that events with durations falling within this threshold are not perceived by the user [23]. Completing task execution earlier than the perception threshold is meaningless since the user will not notice this amount of time and cannot initiate tasks any faster. Therefore, any task shorter than the perception threshold can be potentially executed at a lower performance level, so that its execution time can be stretched up to, but not beyond the perception threshold. The resulting lower power consumption can improve energy efficiency while the user's behavior is unaffected.

To account for all possible users and prevent any potential performance degradation, we assume the lower bound of 50ms as the perception threshold for all users.

In the remainder of this paper, we refer to any task finishing within 50ms as a short task and any task running longer than 50ms as a long task. A short task appears instantaneous to the user even if extended up to 50ms. A long task, however, is perceivable to the user even at the highest performance level. Since the user would not be able to perceive the time difference within 50ms, a long task can be safely prolonged by up to 50ms, assuring that the user's think flow is not interrupted and the subsequent behavior is not affected [3, 20]. To take advantage of the allowed delays in interactive applications, we can potentially put the entire memory subsystem into a low-power state between memory operations and power it up upon a memory access request. The transition latency due to this on-demand power up may slow down a single memory access; nevertheless, from the task perspective, the user may not notice the aggregated delays, as long as the scaled task does not exceed the user's perception threshold as discussed above.

Figure 1 examines the scenario of keeping memory in a low-power state (Low) between accesses for several interactive applications and compares it to the standard system that keeps memory in a high-power state (High). These interactive applications are described in detail in Section 4 of this paper. The tasks from each application are further classified into short tasks and long tasks. Keeping memory in the low-power state can extend task execution beyond the user's perception threshold, as shown in Figure 1 by shaded area in each category. We observe that 93% of short tasks and 58% of long tasks stay within the user's perception tolerance. The longer tasks suffer more since they perform more memory operations and as a result, expose more transition delays. At this point, we can draw two significant observations: (1) there is a tremendous opportunity to save energy within a running application by keeping memory in a low-power state; and (2) some tasks have to be executed with memory in the high performance state, otherwise the degradation would be noticed by the user. The observations justify the need for an intelligent mechanism that is able to accurately identify memory intensive tasks from the majority of low-demand tasks that can be executed at low memory performance.

The majority of tasks in interactive environments are initiated directly by users and the performance demand within an application exhibits a strong correlation to User Interactions (UIs) with the application [1, 4]. In this paper, we will leverage the high-resolution context of user interactions to categorize UI-triggered tasks and correlate their memory behavior with the user interactions. By utilizing this correlation, the proposed mechanisms will select the best memory power states to match the tasks' performance demand.

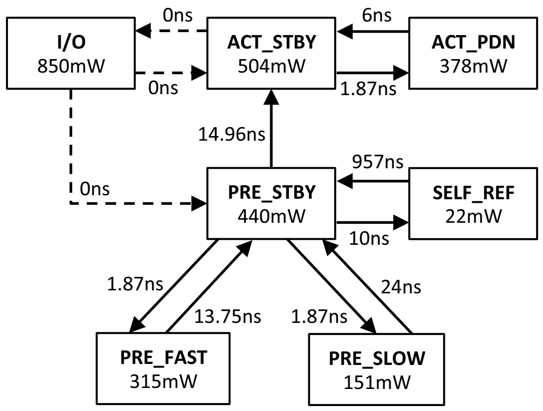
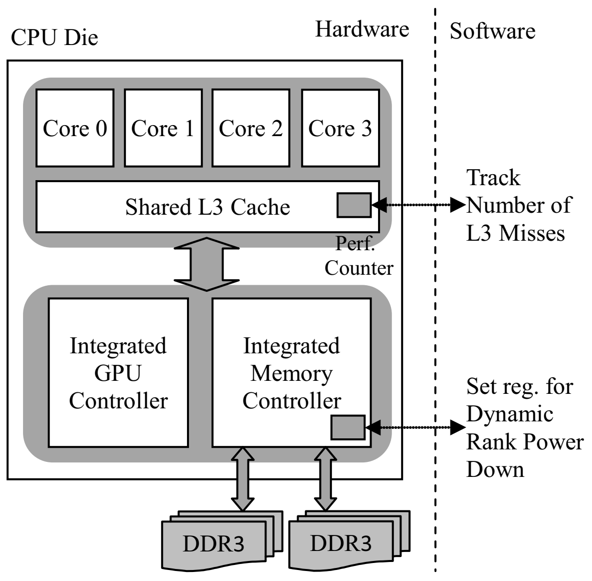


Figure 2: Power specifications for a rank consisting of 8 Micron 1Gbit DDR3-1066 devices.

2.2 Memory Power States

Synchronous Dynamic RAM (SDRAM) is widely used in computers as main memory in the form of Double-Data-Rate (DDR), followed by DDR2 and DDR3. We focus on DDR3 DRAM in this paper since it is the mainstream DRAM architecture in today's computer systems. DDR3 SDRAM is packaged into a DRAM module that commonly consists of two DRAM ranks. The smallest power management unit in DDR3 is the rank and all devices in one rank are operating at the same power state [18]. Each rank in a DRAM module can operate in several different power states: (1) Active Standby state (ACT_STBY): the state where memory can read or write data without any delay; (2) Active Power Down state (ACT_PDN): the power down state that offers some energy savings while minimizing transition delays; (3) Precharge Standby state (PRE_STBY): the intermediate state for transitioning to a much lower energy states; (4) Precharge Power Down Fast (PRE_FAST): the fast power down state where DLL's are still locked; (5) Precharge Power Down Slow (PRE_SLOW): the slow power down state where DLL's are not locked anymore; In both PRE_FAST and PRE_SLOW states several subcomponents of a rank are disabled to reduce power, such as I/O buffers, sense amplifier, row/column decoder, etc.; And finally (6) Self Refresh state (SELF_REF): in addition to previous states, the external clock and on-die termination are disabled to reduce power consumption even further.

Figure 2 illustrates the power specifications for a DDR3-1066 rank [19, 17], including the power consumption, the power state transition, and the associated resynchronization latency. Memory I/Os can only be performed with memory in a high-power state (ACT_STBY); therefore, the rank in low-power states (ACT_PDN, PRE_STBY, PRE_FAST, PRE_SLOW or



CPU Die
Hardware
Software
Core 0
Core 1
Core 2
Core 3
Shared L3 Cache
Perf. Counter
Track Number of L3 Misses
Integrated GPU Controller
Integrated Memory Controller
Set reg. for Dynamic Rank Power Down
DDR3
DDR3

Figure 3: Architecture of Core i CPU.

SELF_REF) has to be transitioned to ACT_STBY state before performing any I/O, potentially exposing a high resynchronization delay to the application. Large number of resynchronizations may cause overall delays long enough to become perceptible to the user and degrade the performance. Therefore, it is critical to control the amount of power state transitions per UI-triggered task.

2.3 Hardware Support

Energy management at the operating system level requires support from hardware to control memory power states and monitor system behavior in detail. Figure 3 shows Intel's most recent Core i CPU with integrated memory controller for low-overhead power state management and monitoring. In addition, the Core i CPU provides a mechanism called Dynamic Memory Rank Power Down to power down memory ranks automatically after a specified memory idle time has elapsed [11]. Subsequently, the operating system only needs to set up the associated register with the desired timeout value to enable this feature. The integrated memory controller will then perform the power state transitions automatically, after the preset timeout expires.

Detailed system monitoring is further provided by the Core i CPU through a set of registers called performance counters, which are crucial for monitoring memory accesses. Under normal operations, virtual memory accesses are invisible to the operating system, while only occasional page faults results in OS being invoked. Performance counters, however, enable the OS to monitor memory activity when applications are performing memory I/Os, such as the number of CPU cache misses that result in main memory accesses. However, exact timing of each memory request, that would allow us to determine memory access burstiness, is not available.

3 IAMEM Design

Observing that there exists a strong correlation between user interactions and the required performance, we propose Interaction-Aware Memory Energy Management for entire memory space in interactive systems. IAMEM will transparently exploit UI events to speculate about the desired performance, and dynamically manage the memory power states to meet the task demand. Subsequently, we will discuss the following components in this section: (1) Unified energy management mechanisms that address all types of accesses to physical memory; (2) High-detail and low-overhead monitoring and detection of tasks triggered by user interactions; (3) Accurate classification and correlation of tasks and the associated processing demand; (4) Online training and prediction for determining the desired memory power for upcoming tasks; and (5) Optimizations to prevent perceivable performance degradation.

3.1 Memory Space

Physical memory in modern operating systems is divided into three categories: (1) kernel space that is strictly reserved for the OS kernel, its data structures, device drivers, etc.; (2) the buffer cache for caching previously accessed disk blocks to improve the file system performance; and (3) user space that is allocated as Virtual Memory (VM) for user processes. The majority of memory space is dynamically allocated to the buffer cache and virtual memory of running processes, based on the current demand for each type of memory.

Memory ranks used for the buffer cache can be efficiently managed in server environments by hiding power-state transition overheads behind the kernel processing time [2]. While the mechanism worked well in server workloads where the buffer cache occupies large space spanning several ranks, it has limited applicability for interactive applications where the buffer cache occupies smaller space and usually shares the rank with the kernel data structures. Subsequently, upon a first kernel memory access, the rank is powered up making large portion of the buffer cache accessible without further delays. Even if the buffer cache occupies several ranks, interactive applications, in general, put lesser pressure on the buffer cache than server applications, such that only a small fraction of overall accesses may require powering up additional ranks. Therefore, we consider memory space occupied by the kernel data structures and the buffer cache as a single kernel memory space and do not distinguish them further. Subsequently, we propose unified energy management mechanisms that manage all memory spaces based on user interaction patterns to guarantee user-perceivable performance while maximizing energy savings.

3.2 UI and Task Monitoring

X Window Server manages interactions with client applications in Linux GUI environment. UI elements are contained in windows organized in a window tree associated with the application. IAMEM relies on a monitoring layer (X Monitor) between the X Server and client applications to uniquely identify individual UI elements [4], as shown in Figure 4. Both keystrokes and mouse interactions with the application are monitored. The unique ID of a given UI element is generated from the element's position within its containing window and that window's position within the window tree. Subsequent interactions with a particular UI element generate the same interaction ID, and the same categorization for the task to follow. The operation of the UI capture mechanism is entirely transparent to the user and does not require any modification to the applications.

Every user interaction results in a task that requires a certain amount of processing to accomplish the goal. The task can be short, such as keystroke capture and display on the screen, or long that involves large amounts of CPU activity, memory I/Os, and even other device I/Os. To reduce the overhead of task detection, IAMEM assumes that a task completes as soon as the OS idle process (swapper process in Linux) begins running and the task is not blocked by I/O, or when the application receives a new UI event [1, 16]. IAMEM uses the Time Stamp Counter to accurately measure the CPU cycles taken to process the task in user and kernel mode, which also include cycles for memory accesses. In addition, IAMEM uses the performance counter to measure the number of accesses to main memory, during task execution, by counting the number of misses in the last level CPU cache.

3.3 UI and Task Correlation

IAMEM classifies tasks by the individual interaction IDs of the triggering UI events. To predict the memory power demand for each task category, IAMEM utilizes a prediction table implemented as a hash table indexed by the interaction IDs. Once the completion of a task is detected, an α aged average method is used to update and record the task processing demand described by the execution time and the number of memory references. The α aged average method captures past behavior of the interaction and also allows quicker adaptation to new

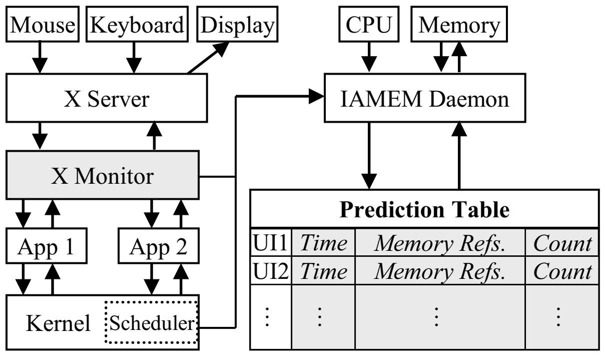


Figure 4: IAMEM design architecture.

behavior patterns, if the memory behavior of the interaction changes. Figure 4 illustrates the prediction table with the following variables for each interaction ID: (1) a weighted sum of all previous tasks' computation time *Time*; (2) a weighted sum of all previous tasks' memory references $MemoryRefs$; and (3) a weighted count of all observed task instances $Count$. For the most recent task with computation time T and memory references M, those three table variables are updated as follows with a predefined weight α ($\alpha <= 1$):

$$Time = \alpha * Time + T \tag{1}$$

$$MemoryRefs = \alpha * MemoryRefs + M \tag{2}$$

$$Count = \alpha * Count + 1 \tag{3}$$

Note that T is recorded as the actual computation time with memory in the active state. When memory transitions occur, the time spent in power state transitions should be deducted from the actually monitored time.

3.4 Power State Prediction

Each time a UI event occurs, IAMEM performs a table lookup using the captured interaction ID. It first predicts the incoming task's processing demand as the average of the retrieved demand history. To maintain the user-perceivable performance, an appropriate deadline D is selected, which is either 50ms for short tasks, or the task's computation time, with memory in the active state, plus 50ms for long tasks. Based on this deadline, IAMEM calculates *PS*, the lowest possible power state of a memory rank, to fit the task execution within the deadline when all predicted memory accesses encounter power state transitions. The prediction algorithm is shown in Figure 5. The resulting *PS* gives us the needed power state to accomplish the task before the deadline and can be any of the five states: Active Standby (ACT_STBY), Active Powerdown (ACT_PDN), Precharge Powerdown Fast (PRE_FAST),

```
$T_{avg} = Time/Count;$
$M_{avg} = MemoryRefs/Count;$

$if\ \ (T_{avg} <= 50ms)$
    $D = 50ms;$
$else$
    $D = T_{avg} + 50ms;$

$if\ \ (T_{avg} + M_{avg} * L_{SELF\_REF} < D)$
    $PS = SELF\_REF;$
$else\ \ if\ \ (T_{avg} + M_{avg} * L_{PRE\_SLOW} < D)$
    $PS = PRE\_SLOW;$
$else\ \ if\ \ (T_{avg} + M_{avg} * L_{PRE\_FAST} < D)$
    $PS = PRE\_FAST;$
$else\ \ if\ \ (T_{avg} + M_{avg} * L_{ACT\_PDN} < D)$
    $PS = ACT\_PDN;$
$else$
    $PS = ACT\_STBY;$
```

Figure 5: The power-state prediction algorithm for an upcoming task. L_{SELF_REF}, L_{PRE_SLOW}, L_{PRE_FAST} and L_{ACT_PDN} are the transition latencies from SELF_REF, PRE_SLOW, PRE_FAST and ACT_PDN state to ACT_STBY state, respectively.

Precharge Powerdown Slow (PRE_SLOW), or Self Refresh (SELF_REF). PRE_STBY is not considered as a viable state for the predictor as ACT_PDN offer better energy efficiency and lower delays than PRE_STBY. PRE_STBY should only be used as an intermediate connecting state when a lower power state (PRE_SLOW, PRE_FAST and SELF_REF) is selected, but not as a steady end state to run the task. Finally, *PS* is set in the memory controller which transitions a rank to the predicted power state after any access to that rank finishes. Therefore, a rank is transitioned to ACT_STBY state upon the first access request and then transitioned to *PS* after that access completes. Once a task completes and the CPU enters an idle state, all ranks are set to SELF_REF state and will remain in that state until a new memory request arrives.

Selection of ACT_STBY state indicates that the running task cannot tolerate any transition delays to finish before the deadline and thus the task must be executed at the highest performance. Selection of low-power states, on the other hand, indicates that the running task can tolerate power-state transition delays associated with the selected power state while still being able to meet the deadline. We should note that the calculations in Figure 5 assume the worst case scenario where memory I/Os are not clustered but arrive one at a time, since we are unable to capture the exact access patterns but only the total number of accesses during the task execution. Subsequently, the calculated delays are the maximum predicted delays the task may encounter, minimizing the possibility that the tasks would continue past the deadline. If memory accesses are bursty, arriving together, the actual exposed delays will be lower.

To avoid exposing potentially large delays to the users while still providing some energy savings, we utilize the ACT_PDN state during the training of the predictor, when the entry in the prediction table is not found. The ACT_PDN state significantly reduces energy consumption as compared to ACT_STBY while keeping the delays low. Finally, interaction IDs are unique across applications and thus can be maintained in a single table in the kernel across executions for all applications, further minimizing the impact of training.

3.5 Improving Prediction Granularity

IAMEM prediction mechanism described earlier utilizes only a single number of memory references to all memory spaces. Once the prediction is made, both user and kernel memory are maintained in the same predicted power state. If this state turns out to be ACT_STBY state, user memory occupied by the given task and the entire kernel memory will be fully powered during the task execution. This behavior may be detrimental to energy efficiency, if for example, the task is computationally intensive with low kernel activity. Subsequently, we extend the design of IAMEM by using two performance counters to monitor references to user and kernel memory individually. We further split MemoryRefs variable in Figure 4 into two fields UserRefs and KernelRefs. We note that user memory and kernel memory including the buffer cache are allocated into separate memory ranks to maximize management efficiency.

Similar to the previous algorithm, a dual prediction algorithm is proposed. In the dual prediction algorithm IAMEM first calculates the average task length T_{avg}, the average number of user memory references M_u and kernel memory references M_k. Then based on the allowed task extension E, IAMEM calculates the lowest combination of power states for user memory PS_u and for kernel memory PS_k, to keep the overall transition delays from both memory spaces below E. Finally, in the case of multiple concurrent threads the thread with the highest demand for memory will dictate the power state for memory.

3.6 Improving Monitoring Accuracy

So far, we have relied on monitoring memory accesses to estimate the worst-case transition overheads for a given task, by assuming that every memory access will require a power state transition. However, some of memory references may arrive in a cluster and encounter one power state transition. To address this issue, we can use another performance counter to count the actual number of power state transitions that a given task encountered. This will allow IAMEM to update the variables of kernel and user memory reference in the prediction table with the actual number of memory references that encountered power state transitions. Subsequently, the algorithms in single and dual prediction remain unchanged except the *M* variables now correspond to the numbers of power state transitions that occurred in user and kernel memory. Utilization of the actual power state transitions accounts for traffic burstiness and as a result, eliminates the inaccuracy resulting from monitoring only memory accesses in the original design.

3.7 Preserving Performance

When a low-power state is predicted for a given task, the accumulated transition delays could become exposed to the user, resulting in performance degradation for longer tasks. The delays can be significant when the long task with high memory activity is mispredicted and the memory is kept in a low power state. To prevent excessive task extension, we propose an early-detect optimization to detect the possible user-perceivable degradation before the task completion. Observing that the scheduler in the kernel will interrupt task execution every 100ms to check if other processes should be scheduled, we add a monitoring module into the scheduler to monitor the given task execution for potentially missed deadlines. Every time the scheduler is invoked, the monitoring module reads the performance counter that counts the number of power state transitions from a low-power state to ACT_STBY state and calculates the actual delay that the current running task has encountered so far. Once the delay exceeds the allowed 50ms extension, the monitoring module will raise the memory power state to the next higher level. If the next higher selection is ACT_PDN state, the monitoring will continue. Seeing an additional delay of 50ms, due to the power state transitions, the memory power state is switched to ACT_STBY for the remaining task execution. This optimization will minimize the potential delays that are exposed to the user.

4 Methodology

We use trace-driven simulation to evaluate the proposed IAMEM and compare it with the following mechanisms:

- **PAVM**. Power-Aware Virtual Memory: The existing state-of-the-art per-process mechanism which keeps the ranks occupied by the currently running process and the ranks occupied by kernel memory in the ACT_STBY state during the process execution, while keeping all other ranks in SELF_REF.
- **ODPD**. On-Demand Powerdown: An existing mechanism that keeps all ranks in the system in the ACT_PDN state during execution and makes transitions to ACT_STBY on memory request arrival. This is the special case of the Dynamic Rank Power Down technique implemented in Intel Core i CPUs, with the idle timeout value set to zero to minimize energy consumption.
- **ODSR**. On-Demand Self Refresh: We propose a complementary mechanism to ODPD that keeps all ranks in the system in SELF_REF and transitions to ACT_STBY upon a memory request arrival.
- **ORACLE**. A per-task mechanism that utilizes the future knowledge to select optimal power states for ranks occupied by user and kernel memory for each incoming task.

Each of the evaluated mechanisms will put all ranks in the system to SELF_REF state when a task completes and the system begins idling. The simulator includes a task scheduler as well as a memory simulator. The memory simulator includes a memory controller and two DDR3-1066 DIMMs, each consisting of two 1GB ranks. The ranks are allocated to minimize fragmentation of memory for running processes across multiple ranks [13]. Finally, the energy management mechanism makes memory power decisions upon each UI event and the memory simulator executes the corresponding power-state transitions for the accessed ranks and calculates energy consumption according to Figure 2.

The application traces used in the simulation were collected using a modified Linux kernel 2.6.30 running on Intel Core i7-920 CPU with 4GB DDR3-1066. All traces contain data of UI events and process activity from a large number of usage sessions in the GNOME environment. UI events were collected with the modified X-Monitor, including the timestamp of each event and the interaction ID uniquely identifying the GUI component. Process activity traces were collected with *Linux Trace Toolkit* that logs program execution details from a patched Linux kernel. Based on the ordered event timestamps, we are able to simulate the dynamic execution progress of the traced applications, so that the computation time for each UI-triggered task can be calculated accurately.

We set up two performance counters: one for counting the event MEM_LOAD_RETIRED.L3_MISS that occurred in user mode, and the other for counting the same event in kernel mode [11] to measure last level (L3) cache misses. Each of the counters was read as soon as an UI event was captured, and was read again upon the completion of the triggered task. The difference of

Traced applications	Trace length	Num.of interaction IDs	Num.of short tasks	Num.of long tasks	Average task length	Average user memory ref.	Average kernel memory ref.
AbiWord	3.1 hrs	179	22840	2080	0.02 sec	7794	2378
Gnumeric	2.9 hrs	342	7804	1416	0.05 sec	19979	5013
Anjuta	3.9 hrs	458	14300	1768	0.03 sec	4127	2457
GIMP	3.3 hrs	228	3356	2440	0.14 sec	39409	13660
LiVES	2.6 hrs	166	2728	1208	0.51 sec	67319	381907
Memtester	0.1 hrs	1	0	20	167.78 sec	264112456	10389729

Table 1: Statistics of application traces.

the two counter values is considered as the number of references in user or kernel memory during this task execution.

We use five commonly executed interactive applications and one benchmark, shown in Table 1: *AbiWord* – a word processing application; *Gnumeric* – a spreadsheet application; *Anjuta* – an Integrated Development Environment for C/C++ and Java development; *GIMP* – an image processing application; *LiVES* – an integrated video editing and playback application; *Memtester* – a memory benchmark that intensively tests the performance of main memory. The traces were collected over a period of several hours and the trace length is shown in the second column. The number of interaction IDs presents the total number of unique user interactions in each application and serves as an indicator of the GUI complexity for the application. In case of *Memtester*, there is only one interaction to start the benchmark. Table 1 also lists the numbers of short tasks (shorter than 50ms), long tasks (longer than 50ms), and the average task length for each application. Finally, the average numbers of memory references in user and kernel memory specifically indicate the per-task demand on the memory subsystem.

4.1 Performance Demand of Applications

Figure 6 shows the distribution of task processing demand for each application based on ORACLE's optimal power selection for user and kernel memory that maximizes energy savings while eliminating delays exposed to the user. *AbiWord*, *Gnumeric*, *Anjuta* and *GIMP* have generally lower performance demand because most of their tasks require users to think to complete interactive operations such as editing text. Subsequently, user memory can stay in either SELF_REF or PRE_SLOW state for a majority of the time during the task execution.

On the other hand, *LiVES* work on large video clips, resulting in significantly higher demand for memory performance. As a result, user memory has to spend more of its time in the higher performance states (PRE_FAST, ACT_PDN and ACT_STBY) to prevent delays from being exposed. Finally, *Memtester* is a memory intensive benchmark that constantly reads and writes the memory and requires the maximum performance. Therefore, user memory must stay in the ACT_STBY state all the time to prevent user perceived delays.

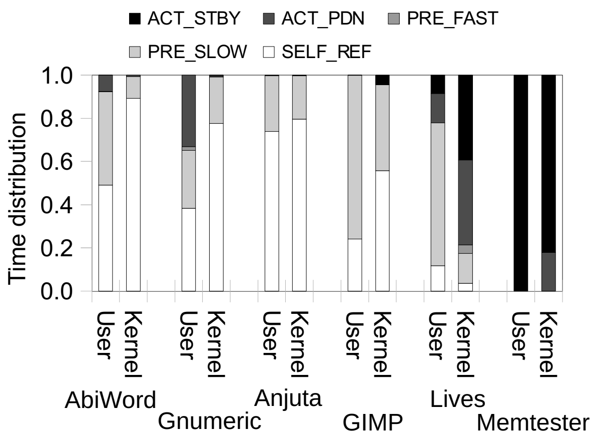


Figure 6: Task performance demand based on the ORACLE's optimal power.

Figure 6 also shows that the performance requirements from kernel memory is lower than user memory. This is because the applications usually invoke few system calls and perform most processing in their own virtual address space, creating lower kernel memory activity as shown in Table 1. Subsequently, kernel memory can stay for 77% of the time in SELF_REF state for *AbiWord*, *Gnumeric*, *Anjuta*, and *GIMP*. However, *LiVES* and *Memtester* demand higher performance from kernel memory and kernel memory must stay in the high power state for the majority of the time to prevent performance degradation.

5 Evaluation

5.1 Energy

Figure 7 shows the average per-task memory energy consumption for each mechanism and application. The energy bars are divided into energy consumed in five power states: ACT_STBY, ACT_PDN, PRE_FAST, PRE_SLOW, and SELF_REF. The energy consumption

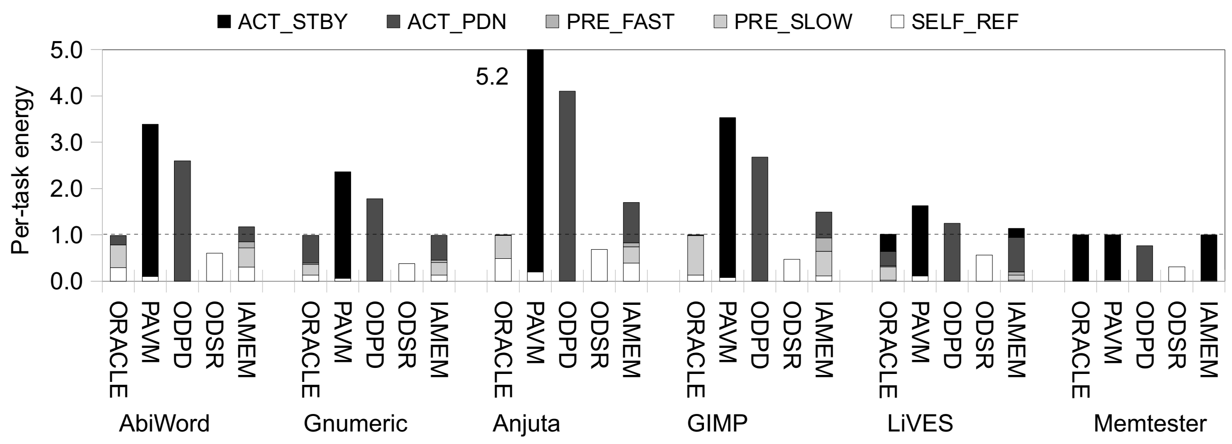


Figure 7: Average per-task memory energy consumption for processing each task normalized to ORACLE.

Applications	ORACLE	PAVM	ODPD	ODSR	IAMEM
AbiWord	1093.05	1503.50	1364.87	1028.45	1105.24
Gnumeric	1119.14	1444.48	1309.74	979.28	1118.74
Anjuta	1276.92	1698.20	1573.95	1245.92	1342.80
GIMP	1262.46	2029.42	1769.18	1100.90	1411.61
LiVES	1648.19	2313.06	1921.93	1202.21	1787.75
Memtester	5022.28	5025.04	3803.16	1547.72	4998.94

Table 2: Total memory energy consumption (in Joules) during the entire application runtime.

of each mechanism accounts for all ranks in the system during the task execution, and is normalized to ORACLE.

PAVM always keeps user and kernel memory in ACT_STBY state during process execution, therefore consuming the most energy, 187% more than ORACLE. ODPD follows PAVM with 119% more energy consumption than ORACLE, because it keeps all memory in ACT_PDN state during the task execution and it is clearly more than necessary for most tasks to eliminate user perceived delays, as show in Figure 6. Additionally, ODPD does not attempt to optimize energy as other mechanisms, which only transition the accessed ranks to the appropriate state. IAMEM closely matches the energy consumption profile of ORACLE through sophisticated demand matching prediction, yielding close to optimal energy efficiency. Subsequently, IAMEM shows less than 14% difference in energy consumption from ORACLE, reducing the energy consumption of PAVM and ODPD by 59% and 47% respectively. In the best case occurring in *AbiWord* and *Anjuta*, where PRE_SLOW and SELF_REF states combined can fit 94% of the time for user memory and 98% of the time for kernel memory, as shown in Figure 6, IAMEM yields as much as 69% improvement in energy efficiency as compared to PAVM and ODPD.

ODSR consumes the least amount of energy when we only consider main memory, yielding 26% less energy consumption than ORACLE, since all ranks are kept in SELF_REF state that has the lowest power demand. However, executing tasks with lower energy consumption than ORACLE is inefficient as it will expose delays to the user and may further increase the energy consumption of the entire system due to the longer runtime. Furthermore, ODSR misses the goal of this paper for transparent energy optimizations that do not expose delays to the user. This scenario also occurs for ODPD in *Memtester*. As shown in Figure 6, *Memtester* requires ACT_STBY state for most of the execution time to avoid performance degradation. However, ODPD utilizes ACT_PDN, and exposes delays to the user. IAMEM still performs almost the same as ORACLE in this case, since it keeps using ACT_STBY state for user memory as required, while recognizing the relatively less demand for kernel memory performance and using the lower ACT_PDN state when necessary.

While Figure 7 shows the energy consumption for processing tasks, it does not reflect the memory energy consumption over the entire execution time since the system idle time is not included. Each of the mechanisms puts all memory ranks in SELF_REF state when the system becomes idle, consuming the same amount of idle energy. Therefore, the overall improvements in energy efficiency originate from the energy savings obtained during the task execution. Table 2 shows the total memory energy consumption during the whole program execution, including the idle time, for each application and each mechanism. As we can see, even considering the total execution time of several hours long (Table 1), IAMEM still gains significant energy savings over PAVM and ODPD, and matches energy consumption of ORACLE with less than 3% difference. For the first five applications, IAMEM consumes 25% and 15% less energy as compared to PAVM and ODPD, respectively. IAMEM energy reduction drops in *Memtester* due to the full power demand from the extremely memory-intensive tasks. Nevertheless, IAMEM still offers slight energy savings as compared to PAVM in this case.

5.2 Performance

While reducing energy consumption is important, preservation of the performance is the goal of this research. Performance degradation can negate any energy savings and negatively affect the user experience. Figure 8 shows the average task length for each application and each mechanism normalized to ORACLE. The task runtime is divided into: 1) task processing time; and 2) the power-state transition time due to the transitions from the lower power state to ACT_STBY state.

We notice that ORACLE introduces some amount of transition time into the task processing time as compared to PAVM that maintains the original task runtime. However, the shorter runtime in PAVM does not translate into better performance since users are not able to notice the shorter task completion and initiate any subsequent interactions. ORACLE always selects the best power state to fit the task execution within the user's perception threshold. The included power-state transition time in ORACLE is not exposed to the user, keeping the user behavior unchanged just like in PAVM. Similarly, ODPD executes most tasks at the higher performance level than desired, resulting in excess energy consumption as shown in Figure 7. Therefore, a task that is executing longer than its execution in ORACLE exposes noticeable delay to the user, while running the task faster is not energy efficient.

Due to the large transition latency (971.96ns) from SELF_REF state, ODSR incurs the most performance degradation, prolonging task execution by 54% on average as compared to ORACLE. *Memtester* exposes the worst case for ODSR with 160% more delay exposed to the user. Memory intensive tasks in *Memtester* result in noticeable delays even in case of ODPD, which only encounters 6ns transition latency for each memory access. IAMEM dynamically recognizes memory intensive tasks and provides appropriate power state similarly to ORACLE, only exposing slightly more than 1% delay to the user for each application. Combining the results from Figure 7 and Figure 8, we observe that IAMEM is the most energy efficient mechanism and almost perfectly matches the behavior of ORACLE. This indicates that utilization of the interaction context allows IAMEM to accurately predict the demand placed on the system and achieve near-optimal energy efficiency without performance degradation.

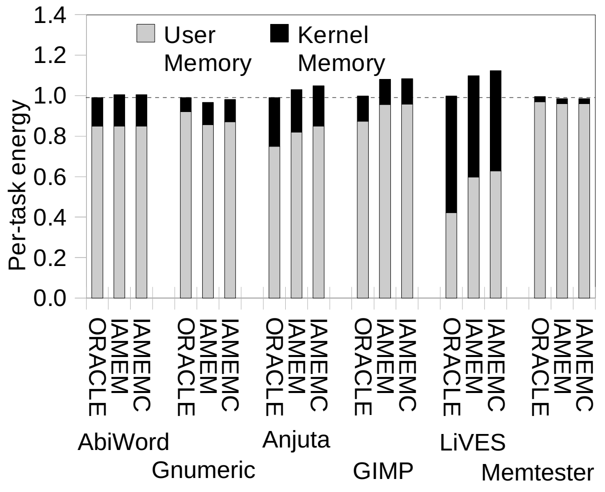


Figure 9: Energy consumption of IAMEM with single and dual prediction.

5.3 The Need for Dual Prediction

Figure 7 showed IAMEM with dual prediction for user memory and kernel memory separately, as described in Section 3.5. Alternatively, IAMEM may also view user and kernel memory as a whole, predicting only a single power state for both memory spaces. Figure 9 compares the average per-task energy of IAMEM with single power prediction (IAMEMC), normalized to ORACLE. This separation allows us to study the contribution of energy consumed by user and kernel memory to the total per-task energy consumption.

As shown in Figure 6, difference in demand for kernel and user performance allows IAMEM to reduce energy consumption in both kernel and user spaces. Therefore, IAMEM reduces the combined energy consumption of user and kernel memory by 3%, on average, as compared to IAMEMC. This benefit of the dual prediction justifies the need for predicting power individually for user and kernel memory.

5.4 Delay Reduction with Early-Detect

Preserving performance and bounding delays exposed to the user is critical for overall energy efficiency and the user's satisfaction. Therefore, IAMEM adopts the early-

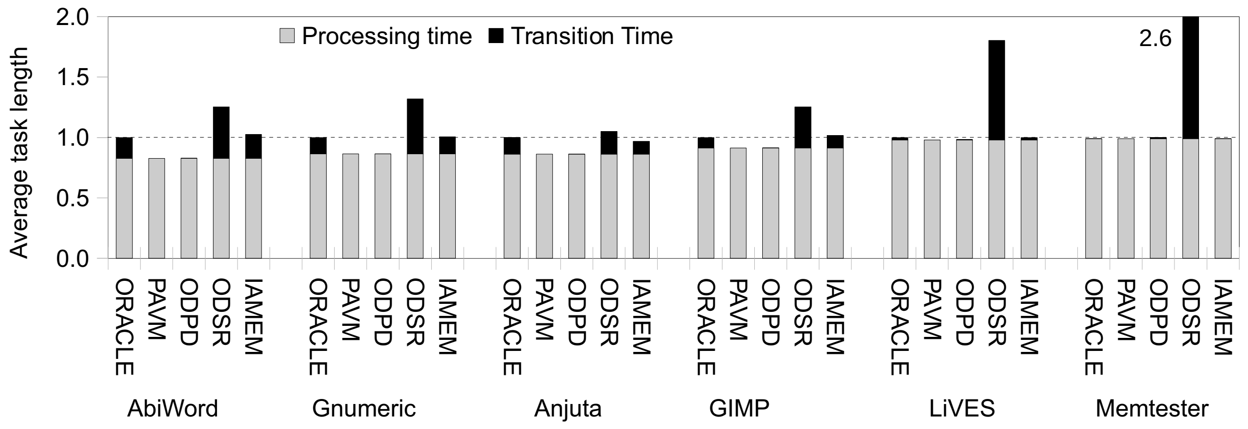


Figure 8: Average per-task length normalized to ORACLE.

detect optimization which uses the scheduler to check every 100ms as discussed in Section 3.7. This optimization will eliminate significant performance degradation for the long running tasks. Table 3 compares the average exposed delays of the long running tasks (longer than the 100ms scheduler interval) for IAMEM with and without this optimization. The delays shown only count the extra times that exceed the allowed 50ms extension and may be noticed by the user. We can see that IAMEM with this optimization significantly reduces the amount of delays exposed to the user for long running tasks. The final small extension above 50ms is at the lower end of the perception threshold range (50-100ms), and will be unnoticeable to the user.

Alternatively, we can remove the optimization since IAMEM without early-detect still manages to keep the delays reasonable that may not be noticed by most users. In addition, IAMEM without early-detect would reduce the per-task energy consumption in Figure 7 by additional 2%, on average. However, the goal of this research is to maximize energy savings without affecting the user experience. The early-detect optimization is critical in achieving this goal, and we subsequently included it in IAMEM implementation and all previous results reflect the inclusion of this optimization.

Application	Num.of long running tasks	Num.of delayed long running	Delay w/ early-detect	Delay w/o early-detect
AbiWord	1808	380	4.5 ms	5.0 ms
Gnumeric	800	220	7.6 ms	9.8 ms
Anjuta	984	120	3.5 ms	13 ms
GIMP	1536	408	13 ms	15 ms
LiVES	760	188	10.8 ms	11.7 ms
Memtester	14	2	51 ms	53 ms

Table 3: Average delays of the delayed long running tasks in standard IAMEM with the early-detect optimization and alternative design without the optimization.

6 Related Work

Energy management for main memory can be implemented in hardware, software, or the combination of both. Hardware level approaches generally utilize the memory controller to monitor the memory traffic as well as the access pattern, and make the power state transitions for specific memory devices based on the observed energy-saving opportunities. Lebeck et al. [12] studied the interaction of page placement with static and dynamic hardware policies to reduce memory power dissipation. The cooperation between the hardware and the OS was also studied in [12]. Subsequently, Pisharah et al. [22] proposed another approach to save memory energy by introducing a hardware called Energy-Saver Buffers to hide the resynchronization costs when reactivating memory modules. Fan et al. [7] further investigated memory controller policies for manipulating DRAM power states in cache-based systems and developed an analytic model that approximates the idle time of DRAM chips using an exponential distribution. Furthermore, observing that significant energy is consumed when memory is actively idle during DMA transfers, Pandy et al. [21] proposed several energy management mechanisms to improve the concurrency level between multiple I/Os to maximize the memory utilization.

Hardware-level energy management may suffer from inaccuracy and may cause unexpected performance degradation. Software-level energy management, on the other hand, can provide more detailed context of execution to make timely power state transitions. Delaluz et al. [5] proposed a compiler-directed approach to cluster the data across memory banks and insert power-state transition instructions into programs by profiling. However, compiler-directed schemes can only work on a single application at a time and demand sophisticated program analysis support. To address the issue, Delaluz et al. [6] also proposed an operating system based solution where the OS scheduler directs the power state transitions by keeping track of accesses for each process in the system. Subsequently, Huang et al. [8] proposed Power-Aware Virtual Memory that manages the power states of memory devices on a per-process basis. A cooperative software-hardware mechanism [10] was further proposed to combine PAVM and the underlying hardware. Huang et al. [9] also proposed memory-reshaping mechanisms that coalesce short idle periods into longer ones through page migration to maximize energy savings. Targeting the buffer cache, Bi et al. [2] utilized the OS I/O handling routines to hide the delays due to memory power state transitions to minimize the impact of aggressive energy management. Finally, Li et al. [15] proposed a mechanism to guarantee the performance by temporarily disabling memory energy management.

7 Conclusion

As current applications are becoming more data-centric, computer systems are equipped with larger capacity and higher performance main memory. As a result, energy consumption of main memory is significantly increasing. In this paper, we addressed interactive systems where most tasks are initiated by the user, and presented the design of IAMEM, a unified approach to manage the energy consumption of the entire memory space. By correlating the memory performance demand to the user interactions, IAMEM is able to accurately select suitable memory power states for UI-triggered tasks, saving energy while preserving the performance of the system. We have shown that compared to the state-of-the-art mechanisms, IAMEM saves 28%-68% of the memory energy consumed for task processing, resulting in up to 16% reduction of the total memory energy consumption during the entire program execution. In addition to the significant energy savings, IAMEM also successfully maintains the user-perceivable performance by hiding delays associated with energy management.

8 Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 0844569.

References

[1] Bi, M., Crk, I., and Gniady, C. Iadvs: On-demand performance for interactive applications. In *HPCA* (2010).

[2] Bi, M., Duan, R., and Gniady, C. Delay-hiding energy management mechanisms for dram. In *HPCA* (2010).

[3] Card, S. K., Robertson, G. G., and Mackinlay, J. D. The information visualizer, an information workspace. In *CHI '91* (New York, NY, USA, 1991), ACM, pp. 181–186.

[4] Crk, I., Bi, M., and Gniady, C. Interaction-aware energy management for wireless network cards. In *SIGMETRICS* (2008).

[5] Delaluz, V., Kandemir, M., Vijaykrishnan, N., Sivasubramaniam, A., and Irwin, M. J. Hardware and software techniques for controlling dram power modes. *IEEE Transactions on Computers 50*, 11 (2001), 1154–1173.

[6] Delaluz, V., Sivasubramaniam, A., Kandemir, M., Vijaykrishnan, N., and Irwin, M. J. Scheduler-based dram energy management. In *DAC* (2002), ACM, pp. 697–702.

[7] Fan, X., Ellis, C., and Lebeck, A. Memory controller policies for dram power management. In *ISLPED '01* (New York, NY, USA, 2001), ACM, pp. 129–134.

[8] Huang, H., Pillai, P., and Shin, K. G. Design and implementation of power-aware virtual memory. In *ATEC '03* (Berkeley, CA, USA, 2003), USENIX Association, pp. 5–5.

[9] Huang, H., Shin, K. G., Lefurgy, C., and Keller, T. Improving energy efficiency by making dram less randomly accessed. In *ISLPED* (New York, NY, USA, 2005), ACM, pp. 393–398.

[10] Huang, H., Shin, K. G., Lefurgy, C., Rajamani, K., Keller, T. W., Hensbergen, E. V., and III, F. L. R. Software-hardware cooperative power management for main memory. In *PACS* (2004), vol. 3471, Springer, pp. 61–77.

[11] Intel. Intel core i7-800 and i5-700 desktop processor series, 2009. Intel Documentation.

[12] Lebeck, A. R., Fan, X., Zeng, H., and Ellis, C. Power aware page allocation. *SIGPLAN Not. 35*, 11 (2000), 105–116.

[13] Lee, M., Seo, E., Lee, J., and Kim, J.-s. Pabc: Power-aware buffer cache management for low power consumption. *IEEE Transactions on Computers 56*, 4 (2007), 488–501.

[14] Lefurgy, C., Rajamani, K., Rawson, F., Felter, W., Kistler, M., and Keller, T. W. Energy management for commercial servers. *Computer 36*, 12 (2003), 39–48.

[15] Li, X., Li, Z., Zhou, Y., and Adve, S. Performance directed energy management for main memory and disks. *Transactions on Storage 1*, 3 (2005), 346–380.

[16] Lorch, J. R. Using user interface event information in dynamic voltage scaling algorithms. In *MASCOTS* (2003), pp. 46–55.

[17] Micron. Ddr3 memory power calculator. Micron Documentation and Support.

[18] Micron. 1gb: x4, x8, x16 ddr3 sdram features, 2009. Micron Documentation and Support.

[19] Micron. Calculating memory system power for ddr3, 2009. Micron Tech Notes.

[20] Miller, R. B. Response time in man-computer conversational transactions. In *AFIPS '68 (Fall, part I)* (New York, NY, USA, 1968), ACM, pp. 267–277.

[21] Pandey, V., Jiang, W., Zhou, Y., and Bianchini, R. Dma-aware memory energy management. In *HPCA* (2006), IEEE Computer Society, pp. 133–144.

[22] Pisharath, J., and Choudhary, A. An integrated approach to reducing power dissipation in memory hierarchies. In *CASES* (New York, NY, USA, 2002), ACM, pp. 88–97.

[23] Schneiderman, B. *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley, 1998.