Scheduler-Based DRAM Energy Management

V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan and M. J. Irwin
The Pennsylvania State University, University Park, PA 16802

ABSTRACT

Previous work on DRAM power-mode management focused on hardware-based techniques and compiler-directed schemes to explicitly transition unused memory modules to low-power operating modes. While hardware-based techniques require extra logic to keep track of memory references and make decisions about future mode transitions, compiler-directed schemes can only work on a single application at a time and demand sophisticated program analysis support. In this work, we present an operating system (OS) based solution where the OS scheduler directs the power mode transitions by keeping track of module accesses for each process in the system. This global view combined with the flexibility of a software approach brings large energy savings at no extra hardware cost. Our implementation using a full-fledged OS shows that the proposed technique is also very robust when different system and workload parameters are modified, and provides the first set of experimental results for memory energy optimization with a multiprogrammed workload on a real platform. The proposed technique is applicable to both embedded systems and high-end computing platforms.

Categories and Subject Descriptors

D.4.0 [**Operating Systems**]: General; D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; B.3.1 [**Memory Structures**]: Dynamic Memory (DRAM)

General Terms

Management, Design, Experimentation

Keywords

Energy Management, DRAM, Operating Systems, Scheduler, Energy Estimation.

1. INTRODUCTION AND MOTIVATION

Optimizing power consumption of computing components has become as important as performance. Reducing the overall energy expended in sustaining computing demands on embedded devices is commonly accepted as being important for conserving battery energy. At the same time, power optimization is also taking center-stage on high end systems due to thermal and packaging considerations. To quote a recent Intel press release [7], 'Power and heat are the biggest issues for this decade'. Energy management is important at different levels of the system architecture. Further, hardware and software power optimization techniques are both equally important, and often go hand-in-hand to complement each other. Adhering to this philosophy, this paper presents a software-based strategy that can be easily incorporated in the operating system, to optimize the energy consumption of DRAMs by exploiting their hardware capabilities of operating in different power modes.

Energy optimization is important for all components including the processor datapath, caches, memory, peripherals, and buses [2]. Optimizing just one may shift the bottleneck to another component. Realizing this, Intel has recently announced a target power budget for each component as a percentage of overall system consumption [6]. Our focus in this paper is on the DRAM energy consumption, whose target consumption in [6] has put a limit of not more than 5% of overall system consumption. It has been observed [2, 8, 16, 18] that the memory system is a dominant consumer of the overall system energy and is an important candidate for software and hardware optimizations. This is especially true for many embedded applications that are typically memory-intensive (i.e., array-dominant such as signal and video processing). In addition, applications are gradually becoming more data-centric with stringent memory requirements (for both storage and speed), causing vendors to incorporate large storage capacities into their offerings. Typically, a computer system contains several DRAM chips (organized in rows/banks and columns), with each of them consuming power even if it is not being currently used. It would be extremely valuable to explore techniques for selectively transitioning the unused memory modules into lower energy consumption modes whenever possible.

As with other hardware components, DRAM modules have started providing power mode control capabilities [13, 14], where individual modules can be selectively transitioned into a low-power operating mode when inactive. There is a performance (and perhaps energy) cost to be paid in exiting this low-power operating mode and coming back to the active mode before a memory request can be serviced (called *resynchronization cost*). In the hope of exploiting these mode control capabilities, there has been recent interest in hardware-based techniques [3, 9] and compiler-directed techniques [3] to explicitly transition individual modules to a low-power operating mode during periods of idleness.

However, each of these approaches has its own limitations. With the compiler-directed approach, we are limited to optimizations for statically analyzable situations, and have to often resort to conservative options. Further, these approaches can reach decisions based on reference patterns for only one program at a time, and do not have information on a more global scale, e.g., across processes running on that system. While we can be much more aggressive/optimistic (perhaps overly optimistic in some circumstances) in hardware-based solutions, there is the cost of the additional hardware that is required to keep track of historical access patterns and to reach decisions based on these patterns.

In this paper, we present an alternate software-based solution, where the transitioning is effected by the operating system (OS).

The operating system has the physical frame usage information across the processes, and can often base its frame allocation policies to each process for energy optimization. Further, it has this information on a global scale, across the processes. This information can be used by the operating system to effect mode transitions at appropriate points when it has a good idea that a module will not be needed for a certain period. For example, if the OS can predict that a given module will not be used by the next process to be scheduled, the module can be transitioned into a low-power operating mode. This prediction can be made through information that can be gleaned by the OS by examining the reference bits in the TLB/page-table. If this turns out to be inaccurate, a resynchronization cost would be paid. Tracking resource access patterns across processes within the operating system has been used in prior research [12] in the context of powering down peripherals mainly, and this is the first investigation of such issues for DRAM power-mode control. Previous OS-based power management studies also considered voltage and frequency scaling [1, 4, 5, 11, 15, 17].

DRAM mode control by the operating system at re-scheduling points attempt optimization at a much coarser granularity than a compiler or a pure hardware based approach, where transitions can potentially be effected at memory access granularities. However, it is our belief (and our experimental results do confirm this) that a substantial portion of the energy savings that those techniques can provide can be attained with our OS scheduler-based mode transition strategy, without requiring extensive compilation support, or any additional hardware support (other than mode control capability). In fact, we go even further to show that if there is additional hardware support such as those that have been proposed in prior research [3, 9], our scheduler-based mode control strategy can be used in conjunction with these schemes to further the energy savings.

Previous studies [3, 9] that have attempted DRAM power optimization have used a simulation-based strategy, using a set of applications running in isolation (dedicated mode). In contrast, this work conducts an evaluation on an actual platform, running a full-fledged operating system with a multiprogrammed workload. This study modifies an off-the-shelf open-source operating system (Linux Redhat 6.2) running on Sun Sparc hardware, to incorporate our energy optimizations. Further, memory references from a multiprogrammed workload running on this system are traced. The Linux scheduler has also been augmented to track the idle times for different modules with our mode-control mechanism. Using several applications on this experimental platform, this paper makes the following contributions:

- We present the first results for the energy savings through power-mode control for a multiprogrammed workload and a real operating system during an actual execution.
- We demonstrate that our scheduler based (software-only) DRAM power mode control strategy can provide the bulk of the energy savings of a pure hardware based approach, without requiring the extra hardware.
- We also show that our scheduler based strategy, can further the energy savings even if a hardware based DRAM energy optimizer is available.

The rest of this paper is organized as follows. Section 2 reviews basic concepts related to DRAM power-mode management and summarizes important characteristics of hardware-based power-mode management strategies. Section 3 presents details of our scheduler-based power-mode management strategy. Section 4 introduces our experimental methodology and presents our results. Section 5 summarizes our major conclusions.

2. MEMORY ARCHITECTURE AND LOW-POWER OPERATING MODES

We focus on an architecture where the memory system is composed of multiple memory modules organized into banks (rows) and columns. Accessing a data in such an architecture would require activating the corresponding modules. There are several ways of saving power in such a memory organization. The approach adopted in this paper is to put the unused memory banks into one of several low-power operating modes. In all our experiments, we use one module per bank; consequently, the terms *bank* and *module* are used interchangeably.

	Energy Consumption (nJ)	Re-synchronization Time (cycles)
Active	3.570	0
Standby	0.830	2
Nap	0.320	30
Power-Down	0.005	9,000

Figure 1: Energy consumption and resynchronization times for different operating modes.

Each memory bank operates independently, and when not in active use, it can be placed into a *low-power operating mode* to conserve energy. Each low-power operating mode works by activating only specific parts of the memory circuitry such as column decoders, row decoders, clock synchronization circuitry, and refresh circuitry (instead of all parts of the circuitry) [13]. The model for our memory system associates an energy consumption value per cycle for four different operating modes together with a resynchronization time that is needed to bring the module back to the active state (which is the only state where a memory request can be serviced). The energy consumptions and resynchronization times for our operating modes are given in Figure 1. Earlier studies [3, 9] have used similar models for the DRAM modules. The energy values shown in this figure have been obtained from the measured current values associated with memory modules documented in memory data sheets (for a 3.3 V, 2.5 nanoseconds cycle time, 8 MB memory) [13]. The re-synchronization times are also obtained from data sheets. Based on trends gleaned from data sheets, the energy values increase by 30% when module is doubled in size.

A bank can be placed into a low-power mode using a hardware-based, software-based, or a hybrid approach [3, 9]. Previous research [3] has shown that hardware-based schemes, particularly a history-based mechanism (called HBP), achieves the best energy savings. To demonstrate that our approach can give energy savings close to those obtained through sophisticated hardware mechanisms, we compare our proposed scheduler-based strategy with two previously proposed hardware based mode control schemes — called Constant Threshold Predictor (CTP) and History Based Predictor (HBP).

Hardware-based mode control mechanisms, in general, monitor the memory references to each DRAM bank, and based on the history, effect appropriate power-mode transitions. The rationale behind CTP is that if a memory bank has not been accessed in a while, then it is not likely to be accessed in the near future (that is, inter-access times are predicted to be long). A (constant) threshold is used to determine the idleness of a bank, and if the next access to this bank does not come within this time, the bank is transitioned to a lower power mode. As in previous studies (e.g., [3]), after 10 cycles of idleness, the corresponding bank is put in standby mode in our experiments. Subsequently, if the bank is not referenced for another 100 cycles, it is transitioned into the napping mode. Finally, if the bank is not referenced for a further 1,000,000 cycles, it is put into power-down mode. Whenever a bank in the low-power mode is referenced, it is brought directly back into the active mode incurring the corresponding re-synchronization costs (based on what mode it was in). HBP, as discussed in [3], estimates the next inter-access time for a bank to be the same as the previous inter-access time. With this estimate, it calculates what would be the ideal power mode that it can transition the bank to after the current access, and at what time it should bring it back to the active mode (so that the processor does not see resynchronization overheads).

3. OUR APPROACH

Adhering to the philosophy that software-based solutions are more flexible, and less expensive, we propose a power-mode control strategy within the OS. The OS not only has physical frame allocation information for each executing process, but also has access to those that are actually being referenced (by sampling the reference bits in the page table/TLB). Further, it has this information not just for one process, but across all processes. Based on this information, it can determine points in the system execution when one or more banks may remain idle for extended periods, and consequently power them down.

We propose performing such power-mode transitions within the OS scheduler, when context switching from one process to the next. With time quantums (the period of time that a process normally runs before being preempted) typically being tens of milliseconds, one can hope to get substantial energy savings if a bank is not used during a time quantum. Also, if a bank is used only by a single process, additional energy savings can be obtained by placing the bank in a low-power mode until the next time quantum allocated to that process. Further, if the degree of multiprogramming increases (i.e., the number of process pseudo-concurrently executing), then the time gap between successive quanta for a process would further go up, increasing the window of opportunity for energy savings with this strategy. If we are able to track what banks are actually being used, then we can expect the same set of banks to be used the next time this application is scheduled. The other banks can be put into low-power mode.

Based on this principle, we describe a scheduler-based power-mode control scheme, that tracks memory banks that are being used by different applications and selectively turning on/off these banks at context switch points.

3.1 Bank Usage Table

The OS tracks memory bank usage of applications with a Bank Usage Table (BUT). Figure 2 shows an abstract view of a BUT which gives the bank usage information for `n` banks (`B1` through `Bn`) and `m` processes (daemons would also be included in this table). An entry in ith row and jth column marked using $\times$ in this table indicates that process i used bank j the previous time it was scheduled. This information can help us in selecting which banks to transition to a low-power mode when the scheduler picks this process the next time. We next discuss how to mark the $\times$ in the BUT.

When a process is selected to run, we put the banks that do not have an $\times$ into *power-down* mode (see Figure 1), and turn on the ones (to *active*) that do. The $\times$ markers for that row are then wiped out. We next need to find out whether for the current quantum, the process actually refers to a given bank. This can be achieved in different ways:

• Whenever the operating system schedules a process, it can wipe out the permissions (read/write/execute) of its virtual pages that have been mapped in (to frames in memory banks) to invalid in the page table, so that any reference to a virtual page by the process would result in a fault (segmentation fault in this case). Note that pages that do not have a physical frame mapping will page fault anyway. At the time of the fault (whether a segmentation or page fault), the OS can update the corresponding column of the BUT based on where the physical frame resides. It can also give back the permission for the remaining virtual pages mapping into the same physical bank so that references to those pages do not fault again in the same time quantum. With this approach, there is at most 1 extra fault per bank — which does not incur the disk overheads but only updates permissions in the page table entry — to track the BUT within each quantum. Further, setting page table entry permissions before scheduling a process is not a significant overhead (e.g., for an 8MB memory bank with 8KB pages, we need to update at most 8MB/8KB (= 1024) entries per memory bank).

• Instead of taking faults, the OS could use reference bits (that are maintained for each virtual page in the page table) to track what is being referenced. Before scheduling a process, the OS could wipe out the reference bits of all pages that have been physically mapped in. When context switching out this process, it could re-examine all these reference bits and mark the corresponding columns of the BUT for whatever banks are referenced. Note that the page replacement algorithm (second chance LRU in Linux) also samples

Process Id	`B1`	`B2`	`B3`	`...`	`Bn`
1		$\times$	$\times$	$\times$	
2	$\times$	$\times$			$\times$
3				$\times$	$\times$
4					$\times$
5	$\times$	$\times$			$\times$
...	...	...	...	...	...
`m`	$\times$		$\times$		$\times$

Figure 2: Bank usage table (abstract view).

these reference bits, resetting them and moving on to the next candidate when searching for the physical frame. This code also needs to be extended to update the BUT when resetting the reference bits. While this approach avoids any extra page faults, it can affect the approximate LRU page replacement algorithm unless that is also made aware of the BUT updates that are taking place.

• The last approach that we identify is actually not an exact mechanism, but tries to approximate bank usage pattern. The advantage with this approach is that it can be much more efficient than the others. In this scheme, when a process is being context switched out, we simply mark the banks in the BUT for which there is an entry referencing a physical frame in that bank in the TLB. If the TLB does a good job of capturing the application's working set, then this scheme is ideally suited for capturing bank access behavior. The only situations when this approach fails to mark all banks being accessed is when the working set is larger, and there is no TLB entry associated with a bank that is currently being used. In this case, TLB misses need to be handled in software (as in our experimental platform)

All these schemes are quite involved, requiring extensive modifications of the operating system. While a comprehensive experimental evaluation of all these approaches is really needed to evaluate their pros and cons, in this study we use the first approach to mark the BUT table entries since our focus here is mainly to demonstrate the potential of our scheduler-based DRAM mode control strategy. All the energy and performance overheads for implementing this approach are reflected in our experimental results.

3.2 Evaluation Strategy

Our current implementation and evaluation platform uses an actual Sun UltraSparc 5, running Linux RedHat 6.2, kernel version 2.2.14. The physical memory available on this machine is 128 MB, and we can give parameters to the system specifying the bank organization. The default number of banks used in our experiments is sixteen (each of which is 8 MB), and the default OS time quantum is 200 milliseconds. However, we also studied the sensitivity of our energy savings to the bank organization.

We have implemented the BUT table maintenance within the Linux kernel by modifying the scheduler and page fault handler. Instead of having a separate global table, we kept each row of the table in the corresponding Linux *task* structure [19] for that process. We also enhanced the page allocation/faulting mechanism to keep track of banks that are being referenced by a process.

As was mentioned earlier, this is the first study to use an actual implementation-based approach for evaluation purposes, considering a full-fledged system operating in a multiprogrammed fashion. While this gives us a very realistic evaluation, one of its drawbacks is the inability to be able to track energy savings/consumption. For our energy calculations, we dump traces of memory references of the system (not just for one application, but across applications) and then use a post-processing strategy to compute memory-energy consumption (based on the memory model discussed earlier). We have ensured that the trace collection facility does not itself perturb the results significantly by using extensive buffers for accumulating logs and flushing them to disk rather infrequently. The memory reference trace contains whether the reference is a load or store, the virtual page numbers, the corresponding physical frame number, and the timestamp of the reference. The timestamp was generated using the high resolution TICK register available on the UltraSparc

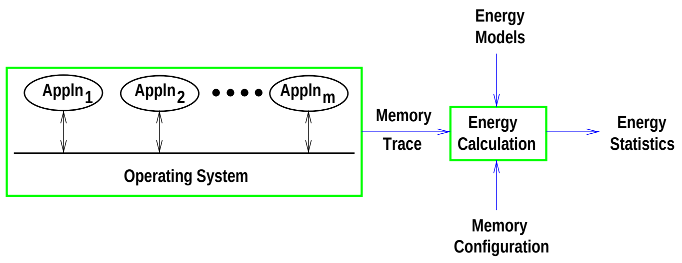

Figure 3: Trace collection and DRAM energy computation.

processor. A schematic of our overall evaluation strategy is given in Figure 3.

3.3 Advantages and Disadvantages

Having presented our approach, we now qualitatively discuss its pros and cons, specifically in relation to hardware-based strategies that have been shown to give good energy savings in prior research [3].

Hardware-based approaches such as CTP or HBP require additional hardware that needs to monitor the ongoing accesses to each bank and then additional logic to make predictions based on this history. This not only incurs additional cost, but may itself consume some amount of energy. Another important concern with hardware-based schemes is that they are not very flexible, and it is not very easy to customize or adapt them based on the current environment/application.

Our scheduler based software strategy, on the other hand, does not require any additional hardware, except the mode transitioning mechanisms that DRAM chips provide for power management. It also offers us the flexibility of determining what banks to power manage, when to initiate transitions, and can adapt to application behavior dynamically. It can also track DRAM usage patterns across applications to base its decisions. For instance, let us say a process that has been accessing certain banks solely dies. The OS can, perhaps, initiate transitions for those banks right away, instead of incurring some delays which are necessary in certain hardware based schemes. In addition, the OS knows when the process is going to be scheduled again after the current epoch. Consequently, to avoid resynchronization costs, it can bring the corresponding banks to active state before the process may need them. Our current implementation does not, however, perform this optimization. An earlier study [12] has also pointed out the benefits of taking actions at the operating system level using task information rather than micro-managing the resource at the device level.

There is a downside to the scheduler-based strategy, which is a consequence of the granularity at which we operate. The mode control decisions are made at milliseconds granularity (at context switch points). On the other hand, hardware-based approaches operate at a much finer resolution, and can effect transitions at an access granularity. However, we think that, in the big picture, extended periods of idleness is what really matters, and our scheduler-based strategy is expected to exploit those periods.

We would like to point out that there is nothing to prevent our scheduler-based approach to be used in conjunction with the hardware-based mechanisms. With such an integrated approach, the scheduler manages the power consumption between one quanta of execution and the next, while the hardware-based mechanism manages the power consumption within each quanta.

We pictorially illustrate the pros and cons of the hardware-based (CTP in particular in this example), scheduler-based and integrated strategies in Figure 4. In this example, we assume two application processes, and only one memory bank being accessed (process 2 does not access this bank during its time quantum). The resynchronizations activities are not explicitly shown for the sake of clarity. The hardware-based strategy activates a bank upon an access, and following a threshold (after the access has been complete) the bank is transitioned into a low-power operating mode. Note that this threshold period could either be completely contained during the quantum for process 1, or some of it may spill over into the quantum for process 2. Note that this may not happen in the scheduler-based strategy since it can put this bank into a low-power mode when process 1 is preempted (if it knows that process 2 does not access this bank). However, it pays the penalty of keeping the bank active during the entire quantum of process 1. The advantage of hardware based vs. scheduler based schemes depends on whether the former or latter factor is more significant. On the other hand, the combined strategy can get the best of both worlds, and in this example gives the maximum energy savings as shown in the figure. This example has been given only to illustrate where each scheme gets its savings, and a realistic evaluation with actual applications is needed to really compare the schemes as is done next.

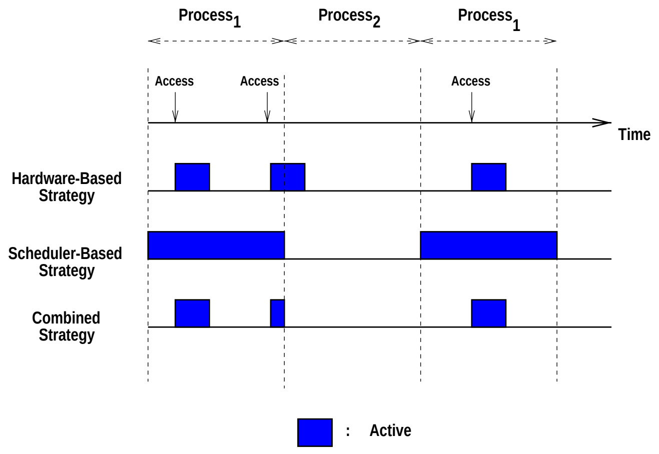

Figure 4: Comparison of hardware-based, scheduler-based, and combined strategies.

Benchmark	Source	Input	Energy (mJ)
rawcaudio	MediaBench	clinton.cpm	123.2
rawdaudio	MediaBench	clinton.adpcm	112.0
polyphase	MediaBench	polyphase.INT polyphase.INT.COE	85.7
md5	MediaBench	-	40.4
cordic	MediaBench	-	44.2
paraffins	Trimaran	15 entries	16.5
g721encode	MediaBench	clinton.pcm	6063.4
mcf	Spec2000	inp.in	3172.4

Figure 5: Benchmark characteristics.

4. EXPERIMENTS AND RESULTS

4.1 Benchmarks

To evaluate the effectiveness of our scheduler-based strategy in saving energy, we used eight complete benchmarks. Six of these benchmarks are from MediaBench [10], a suite of applications that stream data. One of the remaining applications is `mcf`, a representative benchmark (which is very memory-intensive) from the SpecInt 2000 suite and the other is `paraffins` that comes with the Trimaran system. The salient characteristics of these benchmarks are summarized in Figure 5. The last column gives the *memory system energy consumption* for each benchmark when all 16 banks are kept in the fully active (normal operating mode) during the entire execution of the application (assuming no cache memory). All energy numbers reported in the rest of this paper are values normalized to those in the last column.

4.2 Energy Savings and Performance with Single Application Execution

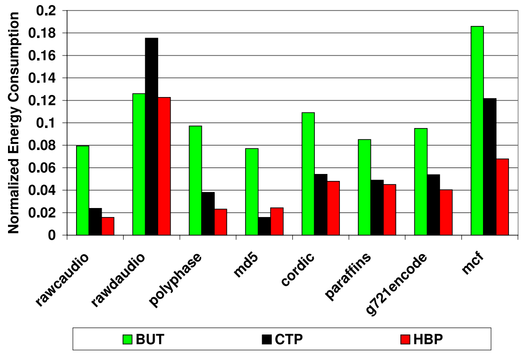

Figure 6: Normalized memory energy consumption.

Figure 6 shows the normalized energy consumption for our benchmarks in an execution environment where only one application is running in the system with no cache memory present (results with a cache are presented later). Recall that there are always some daemons that execute. The energy savings obtained using CTP and HBP are also given. We can observe that CTP and HBP give very good energy savings, as was observed in previous studies. More noteworthy is the observation that our OS scheduler based strategy (denoted as BUT in the figure) is giving most of the energy savings provided by the hardware mechanisms. Please note that while the differences may look deceptively larger, the scale of the graph is quite amplified (for example, BUT gives 92% energy savings for `rawcaudio` while CTP/HBP give around 98-99% energy savings). More importantly, our strategy provides these benefits without incurring any additional hardware costs. This overhead of HBP scheme in particular can account to up to 10% additional energy consumption (consuming 241pJ every cycle) [3]. This overhead is not included in our comparison as it could vary based on the implementation. On the average, we have about 90% savings in DRAM energy with our scheduler-based strategy. We would like to point out that we did not put a bank into low-power mode during the execution of an OS daemon (while the hardware-based mechanisms could). Providing this enhancement can further lessen the differences between the hardware and our software strategies.

As mentioned earlier, we also considered the option of integrating our scheduler-based strategy with the hardware-based mechanism (both CTP and HBP), with the former providing savings across time quantum and the latter providing savings within a time quantum. We find that such a combined strategy can further the energy savings that are obtained with just the hardware-based mechanisms. More specifically, combining CTP with BUT improved the energy consumption of CTP by 22.7% and combining HBP with BUT improved the energy consumption of HBP by 73.0%.

We also found that the performance overhead of our scheduler-based approach is negligible (less than 1% for all benchmarks), assuming a resynchronization latency of 9000 cycles for the power-down mode. In another set of experiments, we increased the resynchronization time ten-fold, anticipating a large relative resynchronization latency for future memory chips (due to increasing microprocessor frequencies). We found that, even under this scenario, the increase in execution cycles is less than 7%.

It should be noted that this set of experiments gives the bias more towards the hardware-based schemes since there is not too much gap between successive time quanta for an application (the only activities between successive quanta for an application is the routine bookkeeping work for the OS). Even in this case, we find that our scheduler-based strategy comes close to the hardware mechanisms. In the rest of the discussion, we focus on our scheduler based strategy and examine its behavior for different workload and hardware parameters.

4.3 Energy Savings with Multiple Applications

We performed another set of experiments where we measured the effectiveness of our strategy under a multiprogrammed workload that imposes a higher demand on the memory system. No previous related study has studied this influence on the effectiveness of the DRAM power management schemes. To conduct this study, we simultaneously ran several instances of the same benchmark in the system. The results are depicted in Figure 7 for `mcf`, with the number of instances varying from 1 to 40. As the multiprogramming level increases, the time gap between successive executions of an application increases, thereby saving more energy (we observed that the OS page allocation provides some amount of insulation between processes by allocating them pages from different banks). However, beyond a certain point (4 in this case), the number of processes becomes so high that their memory demands cause a single bank to be shared by several processes. This is evidenced by the average number of banks used by an instance (process). We found that the average number of banks used at any instance increases from 3.4 when 10 instances are running to 7.6 when 40 instances are running. When an instance is scheduled, the scheduler ends up transitioning fewer banks into the low-power mode. This consequently reduces the energy savings at high multiprogramming levels, making the energy consumption increase almost linearly beyond 10 instances. Nevertheless, we observe that even with forty instances of the application concurrently running, we achieve nearly a 40% saving in memory energy consumption. Similar observations were made when we conducted tests with multiple instances of other benchmarks in our experimental suite.

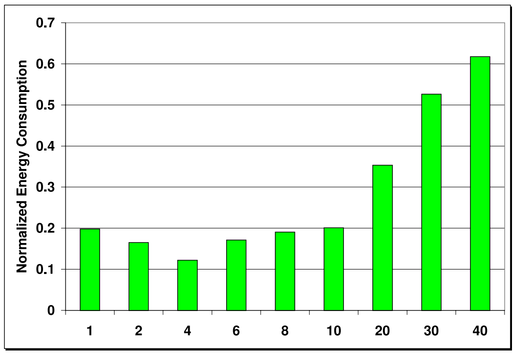

Figure 7: Normalized memory energy consumption (multiple instances of `mcf`).

4.4 Sensitivity to the Number of Banks

In the next set of experiments, we changed the number of banks in the system keeping total memory size fixed at 128MB. Recall that the original bank configuration has sixteen 8MB banks. We observe from Figure 8 that increasing the number of banks improves the energy savings. This is because a larger number of banks allows our approach to manage power-mode transitions for smaller portions of memory (at a finer granularity) and place more memory space into low-power operating mode when inactive. However, our other experiments with different number of banks also reveal that increasing the number of banks beyond a point does not increase the savings further. This is due to the fact that when a certain number of banks is reached, the access pattern of the application does not allow to transition more banks into the low-power mode, i.e., the working set spans that many banks. We also observed that this point is highly application-dependent. Another interesting observation from Figure 8 is that even with only four memory banks, our scheduler-based strategy achieves nearly a 50% saving in memory energy of `mcf`.

4.5 Savings in a Cache-Based Environment

We performed another set of experiments to measure the energy impact of power-mode control in a cache based environment. It should be noted that the behavior of our scheduler-based strategy is oblivious of the cache architecture in the system (if any). Therefore, we report results only for HBP and combined strategies. In

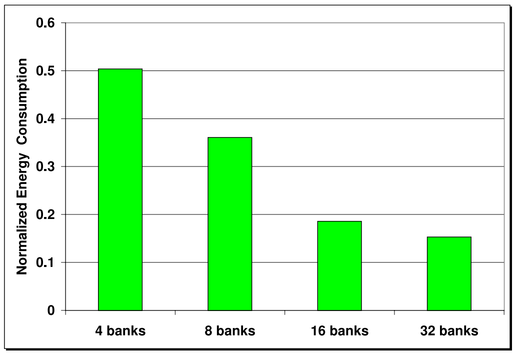

Figure 8: Sensitivity of energy savings to the number of banks (single instance of `mcf`).

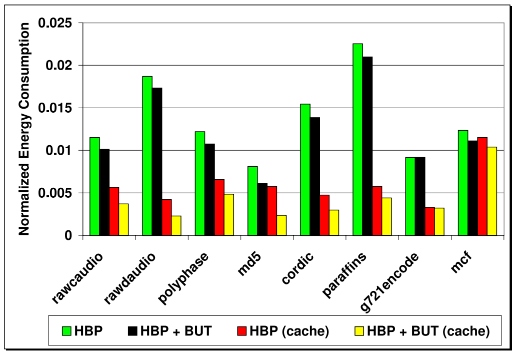

Figure 9: Impact of L1-L2 data cache hierarchy.

this set of experiments, we use a two-way L1 data cache of 16 KB and a direct-mapped L2 cache of 32 KB. The line (block) sizes for L1 and L2 are 32 bytes and 128 bytes, respectively. We present the results in Figure 9. We see from these results that an L1-L2 cache hierarchy can help to improve the energy consumption further, as temporal locality helps the memory banks to transition to more aggressive energy-saving modes (the idleness duration for main memory banks increases). We see that for the `mcf` benchmark the additional savings obtained from the cache are not significant. This is because this benchmark exhibits poor locality, and consequently, has a high miss rate. For the other benchmarks, on the other hand, the improvement ranges from 47% (for `polyphase`) to 80% (for `rawdaudio`). We also observe from these results that even with a cache-based environment, our scheduler-based strategy helps increase the effectiveness of HBP in saving memory energy.

5. CONCLUSIONS

With power/energy optimization taking center-stage together with performance, it is becoming essential to optimize energy consumption at all levels of the system architecture. Such optimizations are not only important in embedded environments, but in high-end systems as well for reliability, packaging, and cooling constraints. Memory energy consumption has been pointed out as playing an important role in many applications, many of which are becoming data centric. At the same time, the small target budget for memory power [6] makes this an important hardware component to optimize using hardware and software techniques.

Earlier research on memory energy optimization has mainly used hardware mechanisms to detect bank idleness, and effect transitions to low-power modes accordingly. Apart from the inflexibility of such mechanisms, hardware cost is an important consideration.

Addressing these drawbacks, this paper has presented a new software mechanism for energy management within the operating system. We have proposed a scheduler-based mode transitioning mechanism for DRAMs that can keep track of banks referenced by different application processes, and use this information to power down banks that will not be referenced in the next time quantum during context switch points. Apart from this new proposal, we have also implemented and evaluated this mechanism on an actual Linux based system using realistic/multiprogrammed workloads. Performance results clearly indicate that our software-based strategy can give most of the savings of more expensive hardware-based schemes. Further, we have also shown that the scheduler-based mode control can even be integrated with an available hardware strategy to further the memory energy savings.

6. ACKNOWLEDGEMENTS

This work was supported in part by NSF grants 0082064, 0093082, 0093085, 0103583, 9701475, 9988164, 0130143 and by GSRC and PDG grants.

7. REFERENCES

[1] L. Benini, A. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8(3), June 2000.

[2] F. Catthoor, S. Wuytack, E. D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology – Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publishers, June 1998.

[3] 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*, November 2001 (Vol.50,No.11).

[4] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proc. the ACM International Conference on Mobile Computing and Networking*, pages 13–25, 1995.

[5] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava. Power optimization of variable voltage core-based systems. In *Proc. Design Automation Conf.*, 1998.

[6] Intel announcement. http://developer.intel.com/design/mobile/intelpower/int_mpg.htm.

[7] Intel announcement. http://www.intel.com/pressroom/archive/releases/20011126tech.htm

[8] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proc. the 37th Design Automation Conference*, Los Angeles, California USA, June 5-9, 2000.

[9] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Proc. ASPLOS'00*, November 2000.

[10] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. In *Proc. the International Symposium on Microarchitecture*, 1997.

[11] J. R. Lorch and A. J. Smith. Scheduling techniques for reducing processor energy use in MacOS. *Wireless Networks*, 3(5):311–324, 1997.

[12] Y.-H. Lu, L. Benini, and G. De Micheli. Operating system-directed power reduction. In *Proc. ISLPED'00*, Rapallo, Italy, 2000.

[13] Rambus Inc. http://www.rambus.com/.

[14] 128/144-MBit Direct RDRAM Data Sheet, Rambus Inc., 1999.

[15] Y. Shin and K. Choi. Power-conscious fixed priority scheduling for hard real-time systems. In *Proc. Design Automation Conference*, pages 134–139, 1999.

[16] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. Y. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using SimplePower. In *Proc. the International Symposium on Computer Architecture*, June 2000.

[17] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proc. Symposium on Operating Systems Design and Implementation*, pages 13–23, 1994.

[18] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The design and use of SimplePower: a cycle-accurate energy estimation tool. In *Proc. the 37th Design Automation Conference*, Los Angeles, California USA, June 5-9, 2000.

[19] M. Beck. *Linux Kernel Internals* (2nd Edition). Addison-Wesley, 1999.