USENIX Association

Proceedings of the General Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003

Design and Implementation of Power-Aware Virtual Memory*

Hai Huang, Padmanabhan Pillai, Kang G. Shin
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
{haih,pillai,kgshin}@eecs.umich.edu

Abstract

Despite constant improvements in fabrication technology, hardware components are consuming more power than ever. With the ever-increasing demand for higher performance in highly-integrated systems, and as battery technology falls further behind, managing energy is becoming critically important to various embedded and mobile systems. In this paper, we propose and implement power-aware virtual memory to reduce the energy consumed by the memory in response to workloads becoming increasingly data-centric. We can use the power management features in current memory technology to put individual memory devices into low power modes dynamically under software control to reduce the power dissipation. However, it is imperative that any techniques employed weigh memory energy savings against any potential energy increases in other system components due to performance degradation of the memory. Using a novel power-aware virtual memory implementation, we estimate a significant reduction in memory power dissipation, from 4.1 W to 0.5–2.7 W, based on Rambus memory specifications, while running various real-world applications in a working Linux system. Unfortunately, due to a hardware bug in the chipset, direct power measurement is currently not possible. Applying more advanced techniques, we can reduce power dissipation further to 0.2–1.7 W, depending on the actual workload, with negligible effects on performance. We also show this work is applicable to other memory architectures, and is orthogonal to previously-proposed hardware-controlled power-management techniques, so it can be applied simultaneously to further enhance energy conservation in a variety of platforms.

1 Introduction

Limiting the energy consumption in mobile/embedded systems such as laptops, personal digital assistants (PDAs) and cellular phones is becoming increasingly important as they become widely used and accepted. With this large user community and a highly competitive market comes the inevitable demand for integrating more features and increased performance into small devices, which, in turn, comes at a cost of increased power dissipation. Products compete based on form factors (smaller and lighter is better) as well as additional features and improved user experience, provided by fast processors, copious memory, resource-demanding software, and power-hungry hardware, thus making energy a precious resource. With hardware continuously improving in performance and price, vendors are able to build systems with higher-performance and higher-power components trying to meet users' ever-increasing demands and compete for customers. However, this results in systems that are over-provisioned with components that provide more capacity, more throughput, and more processing power than needed for the typical workload, and as a result, it is becoming more difficult to maintain long battery life in these devices. To make the situation worse, battery technology is improving at a much slower pace than hardware technology, making the gap between energy supply and demand increasingly larger. To deal with this emerging energy crisis, power management is becoming a more critical task than ever before.

Current hardware technologies allow various system components (e.g., microprocessor, memory, hard disk) to operate at different power levels and corresponding performance levels. Previous research has shown that by judiciously exploiting these power levels, depending on the workload, it is possible to have energy-limited systems built from high-performance and high-peak-power components with a minimal impact on the battery life of the device. The trick is to manage these power levels for each component intelligently based on the actual workload. For idle/normal workloads, some hardware components can be put at lower power levels or even be turned off. On the other hand, during peak workloads, the relevant hardware components are powered up to optimize for performance and provide responsive, high-quality service to users. Workloads on mobile systems such as laptops or PDAs are typically interactive, e.g., text editing, emailing, web surfing, or presenting PowerPoint slides, and due to the slow response time of human users, there are ample opportunities [13] to conserve energy by reducing power levels in various system components without any user-perceived performance degradation. During short intervals of high workload, e.g., switching slides, or recomputing a spreadsheet, the relevant system components can be briefly brought back to the higher-performance/power levels, effectively giving the user the benefits of both low-power

*The work reported in this paper was supported in part by the US Air Force Office of Scientific Research (AFOSR) under grant F49620-01-1-0120.

and high-performance in a single system. However, due to non-negligible transitioning delays of some hardware components, performance/energy may suffer if not handled properly.

A large body of previous research concentrates on reducing the power dissipation of microprocessors due to their high peak-power. Using existing techniques, a Mobile Pentium 4 processor dissipates only 1–2 W on average when running typical office applications despite having a high 30 W peak-power [17]. From a software perspective, further effort to reduce power in microprocessors is likely to yield only a diminishing marginal return. On the other hand, there has been relatively little work done on reducing power used by the memory. As applications are becoming more data-centric, more power is needed to sustain a higher-capacity/performance memory system. Unlike microprocessors, a fairly substantial amount of power is continuously dissipated by the memory in the background independent of the current workload. Therefore, the energy consumed by the memory is usually as much as, and sometimes more than, that of the microprocessor in a system. Implementing Power-Aware Virtual Memory (PAVM) allows us to significantly reduce power dissipated by the memory. In the rest of the paper, we describe our experiences in designing and implementing PAVM in a working system.

Our contributions in this paper are summarized as follows.

- Design of a PAVM system that reduces the overall power dissipation by minimizing the energy footprint of each process in a system.
- Exploration of techniques to reconfigure page allocations dynamically to yield additional energy savings by further reducing per-process energy footprint.
- Use of the Non-Uniform Memory Access (NUMA) techniques, in a novel way, as an abstraction layer to manage memory nodes in reducing power.
- Characterization of the memory usage pattern of processes in a Linux operating system, which allows PAVM to effectively manage all system memory including kernel memory, dynamically-loaded libraries, user process's own private pages, and their interactions.
- Implementation of PAVM in a real, working system (running Linux kernel 2.4.18) to evaluate the effectiveness of our techniques on various SDRAM architectures including SDR, DDR and RDRAM when running real-world applications.

The rest of the paper is organized as follows. Section 2 provides some background information on various memory technologies. Section 3 describes our initial design of PAVM, while Section 4 describes the limitations of this prototype design and the necessary modifications needed to handle the complexity of memory management and task interactions in a real, working implementation. Section 5 presents detailed experimentation results. In Section 6, we discuss the related work. Finally, additional remarks about PAVM and conclusion are given in Sections 7 and 8, respectively.

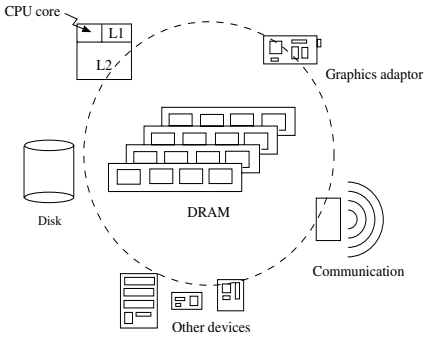


Figure 1: Interaction of memory with the rest of the system.

2 Background

To understand how to reduce power in memory, we must first understand the current memory technologies and the interactions between memory and the rest of the system.

Since 1980, processor speeds have been improving at an annual rate of 80%, while Dynamic Random Access Memory (DRAM) only improved at an annual rate of 7% [39]. Even with a cache hierarchy sitting between the memory and the processor, hiding some of the latencies, the performance gap between the memory and the processor is continuously widening. Memory also interacts with various other system components, such as hard disks, video adapters, and communication devices that use DMA to transfer data as shown in Figure 1. Therefore, memory performance has a significant impact on the overall system performance. Since power reduction is only possible when the memory is operating at lower performance levels, it is critical to implement power-management techniques so that the power reduction in memory justifies any performance degradation, or even power increase, in other system components.

DRAM memory consists of large arrays of cells, each of which uses a transistor-capacitor pair to store a single bit as shown in Figure 2. To counter current leakage, each capacitor must be periodically refreshed to retain its bit information, making memory a continuous energy consumer. Because DRAM fabrication uses advanced process technologies that allow high-capacitance and low-leakage circuits, this refresh occurs relatively infrequently, and is not the largest consumer of DRAM power. Due to the large arrays with very long, highly-loaded internal bus lines, and high degree of parallel operations, significant energy is consumed by row decoders, column decoders, sense amplifiers, and external bus drivers. To reduce power, when a device is not actively being accessed, we can put it into lower power levels by disabling some or all of these subcomponents. However, when it is accessed again, a

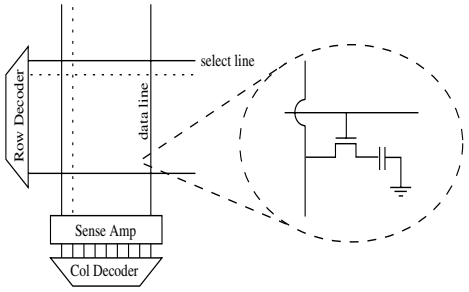


Figure 2: An overview of DRAM architecture with a magnified view of a single DRAM cell composed of a transistor-capacitor pair.

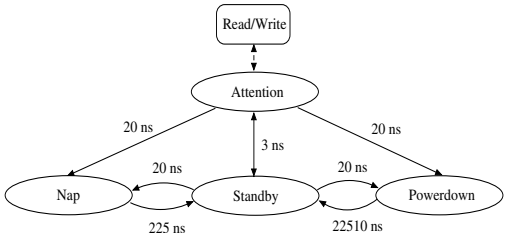


Figure 3: Possible power states for an RDRAM device. The transition time (in nanosecond) between two states is shown on the edge which connects them.

performance penalty is incurred for transitioning from a low-power mode to an active mode by re-enabling these components. This is due to the time needed to power up bus and sense amplifiers, and synchronize with the external clock, so this time penalty is called a *resynchronization cost*. This non-negligible resynchronization cost is the source of performance degradation when power management is not carefully implemented.

The above holds true for all Synchronous DRAM (SDRAM) architectures including the single-data-rate (SDR), the double-data-rate (DDR), and the recently-introduced Rambus (RDRAM) architectures. However, for our energy-conservation purposes, RDRAM differs from the rest by allowing a finer-grained unit of control in power management. In this paper, we consider all three memory types, and show that a finer-grained control can save a significant amount of additional energy over the coarser-grained traditional memory architectures. We now look more closely at these memory architectures with respect to power dissipation.

2.1 SDRAM Architectures

All three types of memory — SDR, DDR, and RDRAM — are physically organized as *modules*, composed of multiple *devices*, each of which is an individually-packaged integrated circuit. The traditional SDR and DDR architectures use wide (64 or 72-bits) data buses at relatively low clock rates (typically, 100 or 133 MHz), and require all devices on the same module to operate in parallel.

In comparison, Rambus DRAM technology [33] transfers data on a narrower 16-bit data channel, operating at twice the clock rate of 400 Mhz, to provide an extremely high throughput to better match the bandwidth needs of modern microprocessors. By using a narrow data bus, only a single device in the module needs to be actively transferring data at a time. This results in a lower power dissipation than for SDR or DDR, where all the devices in the module are activated in parallel to fill the wider bus. Furthermore, since they are not accessed in parallel, the devices in an RDRAM module can be in different power states, giving us a *device-granular* control in power management, in contrast to the traditional SDRAM architectures which can only provide *module-granular* control. Due to this finer-grained level of control, we will primarily focus on Rambus memory, although our approach is also applicable to SDR and DDR architectures, as we will show in Section 5.6.

Power Level	Power	Active Components
Attention	313	Refresh, clock, row, col decoder
Standby	225	Refresh, clock, row decoder
Nap	11	Refresh, clock
Powerdown	7	Refresh

Table 1: Power dissipation (in mW) and active components used for a typical RDRAM device in various power levels.

There are four power levels of interest defined in the RDRAM specification, listed in decreasing order by power dissipation: Attention, Standby, Nap, and Powerdown. Devices are put into lower power levels by disabling the auxiliary subcomponents as discussed previously. For example, while in Attention mode, the self-refresh circuitry and the row/column decoders are active, and the internal clock is kept in-sync with the external clock generator, whereas in Powerdown mode, only the self-refresh circuitry is active to prevent data loss. The details of these power levels and the power dissipation of each are shown in Table 1. However, since read or write operations can only be performed on a memory device when it is in Attention mode, a resynchronization cost is incurred to return a device to Attention mode if it is in any lower power states. The possible state transitions and the corresponding resynchronization costs are shown in Figure 3.

Current RDRAM memory controllers already have a rudimentary form of power-saving policy built-in. Instead of having all devices in Attention mode, the memory controller puts all devices except for the one currently being accessed in Standby mode. Due to the small resynchronization time and the large power difference between Standby and Attention modes, power is significantly reduced with almost no performance loss. Using PAVM, we will show that an additional 59–94% power reduction can be achieved by exploiting the Nap mode with only a negligible performance overhead. Due to the large difference in resynchronization time and the small dif-

ference in power between Powerdown and Nap modes, Powerdown mode is rarely more suitable for use in dynamic power management than Nap mode, as is verified in [9]. So, in this paper, we do not consider Powerdown mode.

3 Design

Prior research on reducing memory power dissipation mainly focuses on power management at a very low hardware level, where memory controllers are responsible for monitoring activity on each memory device and switching devices to lower power states based on various policies for detecting periods of inactivity. This has the benefit of being transparent to the running software, but as the controller is totally unaware of the processes that are using the memory on the system, performing power management at such a low level can often lead to poor decisions at a cost of decreased performance. In this paper, we elevate this decision making to the operating system level, where more information is readily available to make better transitioning decisions to minimize performance degradation and reap greater energy savings.

Before we delve into the design details of PAVM, we first introduce the concept of a memory *node*. We assume that the system memory is partitioned into one or more nodes, where a single node is the smallest unit of memory that can be power-managed independently of other memory. In SDR and DDR, therefore, a node corresponds to a memory module, which contains multiple memory devices, while for RDRAM, it corresponds to a single device within a module. This concept of a node generalizes the unit of control available for performing memory power management operations. We now describe how to manage the nodes to reduce power used by the memory.

3.1 Tracking Active Nodes

Since each node's power level can be separately controlled, total memory power can be reduced by selectively setting nodes to operate at lower power levels. However, selecting which nodes to put into lower power modes is critical to both system performance and power dissipation, since accessing a node in a low-power mode will incur resynchronization costs, stall execution, and, as a result, may increase energy consumption, offsetting any prior savings.

To avoid such costs, we need to ensure that all the nodes a process may access, i.e., its *active nodes*, are kept in high-power state. More specifically, we define a node to be an active node of process i if and only if at least one page from the node is mapped into process i's address space, and we denote the set of active nodes for i as α_i. By promoting the nodes in α_i to Standby mode (high-power) and demoting all other nodes (i.e., those in $\overline{\alpha}_i$) to Nap mode (low-power) when process i is executing, we can reduce power while ensuring that process i suffers no performance degradation due to the increased latency of nodes in low-power states.

Of course, this assumes that α_i can be managed to accurately reflect the active nodes for process i. Previous related research [4] used repeated page faults and page table scans to track the active set, but this involves very expensive, high overhead operations. To track the active set with minimal overheads, for each process we keep an array of counters, each of which is associated with a node in the system. The kernel is modified such that on all possible execution paths in which a page is allocated for, or mapped into, process i's address space, the counter associated with the node containing this page is incremented. Similarly, when a page is unmapped, the counter is decremented. From these counters, α_i is trivially derived: a node is in α_i if and only if process i's counter for the node is greater than zero. The overhead of maintaining α is only one extra instruction per mapping/unmapping operation, and is therefore negligible.

3.2 Reducing Active Set Size

Performing power management based on α's, we can ensure that processes do not suffer any performance losses. However, this does not guarantee energy savings. In particular, if the size of the active set, $|\alpha|$, for each process is close to the total number of nodes in the system, power is not significantly reduced. So, to further reduce power dissipated by the memory, we need to minimize the total number of active nodes used per process for all processes in the system. This can be formally expressed as a minimization problem. Specifically, we want to minimize the summation, $(\sum \omega_i |\alpha_i| : i \in \text{all processes})$, where the number of active nodes, $|\alpha_i|$, for each process i is weighted by its CPU utilization (fraction of processing capacity/time spent executing the process), denoted by ω_i. Allocating pages for all processes among the nodes to minimize this sum is a difficult problem even with a static set of tasks, let alone in a dynamic system.

For simplicity, we assume that an approximate solution can be obtained by minimizing the number of active nodes for each process. To this end, a simple heuristic can be applied using the concept of a *preferred node* and maintaining a set of preferred nodes, ρ_i, for each process i. All processes start with an empty set ρ. When a process i allocates its first page, this page is taken from the node with the most free memory available, which is then added to ρ_i. Future memory allocations by this process are first tried on nodes in ρ_i. If all nodes in ρ_i are full, the allocation is again made from the node which currently has the most free memory available, and this new node is then added to ρ_i. By using this worst-fit algorithm to generate ρ, each process's memory footprint is packed into a small number of nodes, thereby decreasing each process's energy footprint.

3.3 A NUMA Management Layer

Implementing PAVM based on the above approaches is not easy on modern operating systems, where virtual memory (VM) is extensively used. Under the VM abstraction, all processes and most of the OS only need to be aware of their own virtual address spaces, and can be totally oblivious to the actual physical pages used. Effectively, the VM decouples page allocation requests from the underlying physical page allocator, hiding much of the complexities of memory management from the higher layers. Similarly, the decoupling of layers works in the other direction as well — the physical page allocator does not distinguish from which process a page request originates, and simply returns a random physical page, treating all memory uniformly. When performing power management on memory nodes, however, we cannot treat all memory as equivalent, since accessing a node in low-power state will incur increased latencies and overheads, and the physical memory address of allocated pages critically affects each process's energy footprint. Therefore, we need to eliminate this decoupling and make the page allocator conscious of the process requesting pages, so it can nonuniformly allocate pages based on ρ_i to minimize $|\alpha_i|$ for each process *i*.

This unequal treatment of sections of memory due to latencies and overheads for access is not limited to power-managed memory. Rather, it is a distinguishing characteristic of Non-Uniform Memory Access (NUMA) architectures, where there is a distinction between low-latency local memory and high-latency remote memory. In a traditional NUMA system, the notion of a *node* is more general than what we defined previously and can encompass a set of processors, memory pools, and I/O buses. The physical location of the pages used by a process is critical to its performance since intra- and inter-node memory access times can differ by a few orders of magnitude. Therefore, a strong emphasis has been placed on allocating and keeping the working set of a process localized to the local node.

In this work, by considering a node simply as a section of memory with a single common access time, for which the power mode can be set independently of other nodes, we can employ a NUMA management layer to simplify the nonuniform treatment of the physical memory. With a NUMA layer in place below the VM system, physical memory is partitioned into multiple nodes. Each node has a separate physical page allocator, to which page allocation requests are redirected by the NUMA layer. The VM is modified such that, when it requests a page on behalf of process *i*, it passes a hint (e.g., ρ_i) to the NUMA layer indicating the preferred node(s) from which the physical page should be allocated. If this optional hint is given, the NUMA layer simply invokes the physical page allocator that corresponds to the hinted node. If the allocation fails, ρ_i must be expanded as discussed previously. By using a NUMA layer, we can implement PAVM with preferential node allocation without having to re-implement complex low-level physical page allocators.

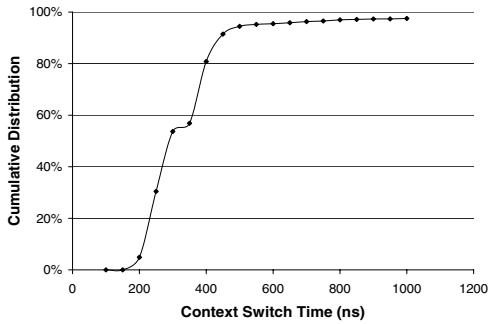


Figure 4: Cumulative distribution of context switch times.

3.4 Hiding Latency

Although the methods discussed so far ensure that a process experiences no performance loss during its execution, there still remains the issue of resynchronization latencies when transitioning power states of the nodes before a process is executed. As mentioned earlier, with RDRAM, switching a device from Nap to Standby mode requires 225 ns, which is not a very long time, but is nontrivial, as it would be incurred on every context switch. If this latency is not properly handled and hidden, it could, as a result of increased runtimes, erode the energy savings and undermine the techniques described above.

One possible solution is that at every scheduling point, we find not only the best process i to run, but also the second best process j. Before making a context switch to process i, we transition the union of the nodes in α_i and α_j to Standby mode. The idea here is that with a high probability, at the next scheduling point, we will either continue execution of process i or switch to process j. Effectively, the execution time of the current executing process will mask the resynchronization latency for the next process. Of course, the cost here is that more nodes need to be in Standby mode than needed for the current process, incurring greater energy costs, but, with a high probability, performance degrading latencies are eliminated.

A second solution is more elegant, has lower computational and energy overheads, and uses the context switching time to naturally mask resynchronization latencies. This is based on the fact that context switching takes time, due to loading new page tables and modifying internal kernel data structures, even before the next process's memory pages are touched. We instrumented the Linux 2.4.18 kernel to measure the portion of context switching time in the scheduler function after deciding which process to execute next, but before beginning its execution. The cumulative distribution of context switch times on a Pentium 4 processor clocked at 1.6 GHz is shown in Figure 4. From the figure, we can see that over 90% of all context switches take longer than 225 ns, and therefore, can fully mask the resynchronization latency for transitioning nodes from Nap to Standby mode. The sharp increase in the cumulative distri-

bution function between 175 and 225 ns indicates that we only pay a few tens of nanoseconds for the other less than 10% of context switches. This approach, therefore, hides most of the latency without incurring additional energy penalties.

With faster processors in the future, the cumulative distribution function of context switch times shifts left, making the second solution less attractive as the latencies will less likely be masked. The first solution proposed is more general and may be applied without any hardware constraints, but at a higher energy overhead. In reality, however, even as processor frequencies are rapidly increasing, context switch times improve rather slowly, so the second solution is viable under most circumstances.

4 Implementation

In this section, we describe our experiences in implementing and deploying PAVM in a real system. Due to complexities in real systems, a direct realization of the PAVM design described earlier does not perform up to our original expectations. Further investigation into how memory is used and managed in the Linux operating system reveals insights that we use to refine our original system and succeed in conserving a substantial amount of memory energy under the complex real-world environments.

4.1 Initial Implementation

Our first attempt to reduce memory power dissipation is a direct implementation of the PAVM design described in Section 3 within the Linux operating system. We extend the task structure to include the needed counters to keep track of the active node set, α_i, for each process i. As soon as the next-to-run process is determined in the scheduling function, but before switching contexts, the nodes in α of that process are transitioned to Standby mode, and the rest are transitioned to Nap mode. This way, power is reduced, the resynchronization time is masked by the context switch, and the process does not experience any performance loss.

We also modify page allocation to use the preferred set, ρ, to reduce the size of the active sets. Linux relies on the buddy system [20] to handle the underlying physical page allocations. Like most other page allocators, it treats all memory equally, and is only responsible for returning a free page if one is available, so the physical location of the returned page is generally nondeterministic. For our purpose, the physical location of the returned page is not only critical to the performance but also to the energy footprint of the requesting process. Instead of adding more complexity to an already-complicated buddy system, a NUMA management layer is placed between the buddy system and the VM, to handle the preferential treatment of nodes.

The NUMA management layer logically partitions all physical memory into multiple nodes and manages memory at a node granularity. The Linux kernel already has some node-specific data structures defined to accommodate architectures with NUMA support. To make the NUMA layer aware of the nodes in the system, we populate these structures with the node geometry, which includes the number of nodes in the system as well as the size of each node. As this information is needed before the physical page allocator (i.e., the buddy system) is instantiated, determining the node geometry is one of the first things we do at system initialization. On almost all architectures, node geometry can be obtained by probing a set of internal registers on the memory controller. On our testbed with 512 MB of RDRAM, we are able to correctly detect the 16 individual nodes, each consisting of a single 256 Mbit device. Node detection for other memory architectures can be done similarly.

Unfortunately, NUMA support for x86 in Linux is not complete. In particular, since the x86 architecture is strictly non-NUMA, some architecture-dependent kernel code was written with the underlying assumption of having only a single node. We remove these hard-coded assumptions and add multi-node support for x86. With this, page allocation is now a two-step process: (i) determine from which node to allocate, and (ii) do the actual allocation within that node. Node selection is implemented trivially by using a hint, passed from the VM layer, indicating the preferred node(s). If no hint is given, the behavior defaults to sequential allocation. The second step is handled simply by instantiating a separate buddy system on each node.

With the NUMA layer in place, the VM is modified such that with all page allocation requests, it passes ρ of the requesting process down to the NUMA layer as a hint. This ensures that allocations tend to localize in a minimal number of nodes for each process. In addition, on all possible execution paths, we ensure that the VM updates the appropriate counters to accurately bookkeep α and ρ for each process with minimal overheads, as discussed in Section 3.

4.2 Shared Memory Issues

Having debugged the new implementation, and ensured the system is stable with the new page allocation method, we evaluate PAVM's effectiveness at reducing energy footprints of processes. We expect that the active node set, α, for each task will tend to localize to the task's preferred node set, ρ. However, this is far from what we see.

Table 2 shows a partial snapshot of the processes in a running system, and, for each process i, indicates the nodes in sets ρ_i and α_i, as well as the number of pages allocated on each node.[1] It is clear from the snapshot that each process i has a large set of active nodes, where $|\alpha_i|$ is much larger than the corresponding $|\rho_i|$. This causes a significantly larger energy

[1] We only show a partial list of processes running in the system due to space limitation, but other processes behave similarly.

Process	ρ	α							
syslog	**14**	**0**(3)	**8**(5)	**9**(51)	**10**(1)	**11**(1)	**13**(3)	**14**(76)	
login	**11**	**0**(12)	**8**(7)	**9**(112)	**11**(102)	**12**(5)	**14**(20)	**15**(1)	
startx	**13**	**0**(21)	**7**(12)	**8**(3)	**9**(7)	**10**(12)	**11**(25)	**13**(131)	**14**(43)
X	**12**	**0**(125)	**7**(23)	**8**(47)	**9**(76)	**10**(223)	**11**(19)	**12**(1928)	**13**(82)
		14(77)	**15**(182)						
sawfish	**10**	**0**(180)	**7**(5)	**8**(12)	**9**(1)	**10**(278)	**13**(25)	**14**(5)	**15**(233)
vim	**10,15**	**0**(12)	**9**(218)	**10**(5322)	**14**(22)	**15**(4322)			
...	...	...							

Table 2: A snapshot of processes' node usage pattern using the initial version of PAVM. The number in parenthesis besides each active node indicates the number of pages the corresponding process is currently using on that node. Recall that our system has 16 nodes, denoted as 0, 1, ..., 15, each contains 256 Mbits (or 8192 4-KB pages).

footprint for each process than what we have originally anticipated. Nevertheless, since most pages are allocated in the preferred nodes, and none of the processes use all nodes in the system, we still consider this a working system that provides opportunities to put nodes into low-power modes and conserve energy. However, it is not as effective as we would like, due to the fact that for each process, there is a set of pages scattered across a large number of nodes.

To understand this "scattering" effect, we need to investigate how memory is used in the system. In most systems, a majority of the system memory is occupied by user processes. In turn, most of these pages are used to hold memory-mapped files, which include binary images of processes, dynamically-loaded libraries (DLL), as well as memory-mapped data files. To reduce the size of the executable binaries on disk and the processes' cores in memory, DLLs are extensively used in Linux and most other modern operating systems. The scattering effect we observe is a result of the extensive use of DLLs combined with the behavior of the kernel-controlled page cache.

The page cache is used to buffer blocks previously read from the disk, so on subsequent accesses, they can be served without going to the disk, greatly reducing file access latencies. When a process requests a block that is already in the page cache, the kernel simply maps that page to the requesting process's address space without allocating a new page. Since the block may have been previously requested by any arbitrary process, it can be on any arbitrary node, resulting in an increased memory footprint for the process. Unfortunately, this is not limited to shared data files, but also to DLLs, as these are basically treated as memory-mapped, read-only files. The pages used for DLLs are lazily loaded, through demand paging. So, when two processes with disjoint preferred nodes access the same library, the pages will scatter across the union of the two preferred node sets, depending on the access pattern of the processes and which process first incurred the page-fault to load a particular portion of the library into the page cache.

In the following sections, we describe the incremental changes we make to reduce the memory/energy footprint for each process by using DLL aggregation and page-migration techniques. We then discuss how to reduce overhead of these new techniques.

Process	ρ	α							
syslog	**14**	**0**(108)	**1**(2)	**11**(13)	**14**(17)				
login	**11**	**0**(148)	**1**(4)	**11**(98)	**15**(9)				
startx	**13**	**0**(217)	**1**(12)	**13**(25)					
X	**12**	**0**(125)	**1**(417)	**9**(76)	**11**(793)	**12**(928)	**13**(169)	**14**(15)	
sawfish	**10**	**0**(193)	**1**(281)	**10**(179)	**13**(25)	**14**(11)	**15**(50)		
vim	**10,15**	**0**(12)	**1**(240)	**10**(5322)	**15**(4322)				
...	...	...							

Table 3: Effect of aggregating pages used by DLLs.

4.3 Revision #1: DLL Aggregation

Due to the many benefits of using dynamically-loaded libraries (e.g., libc), most, if not all processes make use of them, either explicitly or implicitly. Therefore, a substantial number of pages within each process's address space may be shared through the use of DLLs. As discussed above, this sharing inevitably causes pages to be littered across memory, resulting in a drastic size-increase of α_i for each process i.

The cause of this scattering effect is that we are trying to load library pages into the preferred nodes of processes which initiated read-in from disk, as if these were private pages. To alleviate the scattering effect on the library pages, we need to treat them separately in the NUMA management layer. We implement this simply by ignoring the hint (ρ) that is passed down from the VM layer, and instead, resorting to a sequential first-touch policy, where we try to allocate pages linearly starting with node 0, and fill up each node before moving onto the next node. This ensures that all DLL pages are aggregated together, rather than scattered across a large number of nodes. Table 3 shows a snapshot of the same set of processes under the same workload as in Table 2, but with DLL aggregation employed.

As expected, aggregating DLL pages reduces the number of active nodes per process. However, a new problem is introduced. Due to the extensive use of DLLs, by grouping pages used for libraries onto the earlier nodes, we allocate a large number of pages onto these nodes and quickly fill them. As a result, processes need several of these low address nodes in their active sets to access all of the needed libraries. In the two snapshots shown, this is clearly apparent: after aggregation (Table 3), both nodes 0 and 1 are mapped in all of the process active sets, whereas only node 0 was needed without aggregation (Table 2). With many libraries loaded, we would use up these earlier nodes fairly quickly, and may increase the memory footprint of processes. We explain this in more details in the next section and also describe how to alleviate the extra burden on these earlier nodes.

4.4 Revision #2: Page Migration

Even after aggregating library pages, there is still some scattering of pages across nodes outside of ρ for each process. Some of this is due to actual sharing of pages, but the rest is due to previous sharing and residual effects of past file accesses in the page cache. Furthermore, even though aggregating all li-

Process	ρ	α			
syslog	**14**	**0**(15)	**14**(125)		
login	**11**	**0**(76)	**11**(183)		
startx	**13**	**0**(172)	**13**(82)		
X	**12**	**0**(225)	**1**(2)	**12**(2220)	
sawfish	**10**	**0**(207)	**1**(56)	**10**(436)	
vim	**10,15**	**0**(12)	**1**(240)	**10**(5322)	**15**(4322)
...	...	...			

Table 4: Effect of library aggregation with page migration.

brary pages ensures shared pages are kept in a few nodes, not all libraries are shared, or remain shared as the system execution progresses. It is better to keep these pages in the preferred nodes of the processes that are actively using them, rather than polluting nodes that are used for library aggregation and increasing the energy footprints of all processes. We can address all of these by using page migration.

In NUMA systems, page migration is used to keep the working set of a process local to the execution node in order to reduce average access latency and improve performance, particularly when the running processes are migrated to remote nodes for load-balancing purposes. In the context of PAVM, there is no concept of process migration, or remote and local nodes, but we can use the page-migration technique to localize the working set of a process to a fewer number of nodes and overcome the scattering effect of shared pages and items in the page cache. This will allow us to have more nodes in low-power states, thereby conserving more energy.

In our implementation, page migration is handled by a kernel thread called *kmigrated* running in the background. As with other Linux kernel threads, it wakes up periodically (every 3 seconds). Every time it wakes up, it first checks to see if the system is busy, and if so, it goes back to sleep to avoid causing performance degradation to the running processes. Otherwise, it scans the pages used by each process and starts migrating pages that meet certain conditions. We further limit any performance cost by setting a limit on the number of pages that may be migrated at each invocation of *kmigrated* to avoid spikes in memory traffic. Effectively, by avoiding performance overheads, we only pay a fixed energy cost for each page migrated.

A page is migrated if any of the following conditions holds.

- If a page is a process's *private* page (i.e., is used only by that process), and it is not on a node in that process's preferred set, ρ, then the page is migrated to any node in ρ. This will not affect the size of the active set, α, of other processes.
- If a page is *shared* between multiple processes, and the node that it resides on is outside of at least one of these processes' preferred sets (i.e., $\notin \bigcap \rho_i$), then the page is migrated to an earlier node so it can be aggregated with the other shared pages, if and only if this migration does not cause the size of α to increase for any of the processes sharing the page.

Migrating a process's private page is straightforward. We simply allocate a new page from any node in ρ of that process, copy the contents from the old page to the new page, redirect the corresponding page table entry in that process's page table to point to the new page, and finally free the old page.

Migrating a shared page is more difficult. First, from the physical address of the page alone, we need to quickly determine which processes are sharing this page so we can check if it meets the migration criterion given above. Second, after copying the page, we need a quick way to find the page table entry for each of the sharing processes, so we can remap the entries to point to the new page. If any of the above two conditions cannot be met, an expensive complete scan of the page tables of all processes is needed for migrating each shared page. Unfortunately, in the default Linux 2.4.18 kernel, neither requirement is met.

To our aid, Van Riel [34] has recently released the *rmap* kernel patch, a reverse mapping facility that meets both requirements nicely, and is included in the default kernel of the RedHat 7.3 Linux distribution. With *rmap*, if a page is used by at least one process, it will have a chain of back pointers (*pte_chain*) that indicates all page table entries among all processes that point to this page (meets the second requirement). In turn, for each page table containing the above page table entries, there is a back pointer indicating the process that uses this page mapping, satisfying the first requirement. So, when trying to migrate a shared page, we first allocate a new page, and find all the processes sharing this page to determine whether migrating this page will cause memory footprint to increase for any of the processes. If not, we copy the contents from the old page to the new page, replace all page entries that point to the old page with ones pointing to the new page, update the reverse mappings in the new page, and finally free the old page.

With *kmigrated* running, processes use much fewer nodes than in the initial version of the implementation, as shown in the snapshot in Table 4. In turn, memory power dissipation is significantly reduced for each process. However, for each page migrated, we incur a fixed energy cost for performing the memory-to-memory copies.

4.5 Revision #3: Reducing Migration Overhead

Although page migration greatly reduces the energy footprints of processes, it triggers additional memory activity, which may undermine the energy savings obtained. Thus, we must consider ways to limit the actual number of migrations to keep its benefits without incurring too much of energy cost. In this section, we propose two solutions to reduce the number of page migrations.

Solution 1: The DLL aggregation technique described pre-

Application	Interval	Light	Poweruser	Multimedia	Description
X+GNOME	continuous	x	x	x	runs X server using the default GNOME desktop environment
Mozilla	15 seconds	x	x		retrieves and displays webpages from randomly pre-generated URLs
XMMS	continuous	x	x		plays a stream of mp3 files
text editing	60 seconds	x	x		modifies a tex file, runs latex, bibtex, dvips, and displays it in ghostview
gcc	10 minutes		x		compiles Linux-2.4.18 kernel and kernel modules
Xine	continuous			x	plays an MPEG4-encoded movie in full-screen mode

Table 5: Description of the applications used in Light, Poweruser and Multimedia workloads.

viously assumes libraries tend to be shared. Any library that is not shared will later be migrated to the process preferred nodes. This is not efficient for those applications that use proprietary dynamic libraries. We can keep track of the processes that cause a large number of page migrations, and then classify them further as *private-page dominated* and *shared-page dominated*. A process is *private-page dominated* when the number of private pages migrated is much larger than the migrated shared pages. It indicates that the pages this process uses are less likely to be shared, meaning that we should allocate pages on this process's preferred node and not automatically aggregate the library pages it uses.

On the other hand, if a process is *shared-page dominated*, it means that many shared pages were wrongfully migrated initially and later migrated back. For these processes, we want to inhibit the number of page migrations for shared pages to prevent future migrations to correct the initial migration decision.

Solution 2: It is widely known that processes are short-lived. Process lifetime is similar to what is shown in Figure 5 [27], where only 2% of all processes live more than 30 seconds. Instead of performing page migration for all processes, we only migrate pages on behalf of long-lived processes, since the energy spent on migrating pages for short-lived processes does not justify the resulting energy savings. Note that the implementation of *kmigrated* implicitly avoids migrating all processes, as it checks the system at most once every 3 seconds, and only when the system is not busy, thus avoiding most short-lived processes.

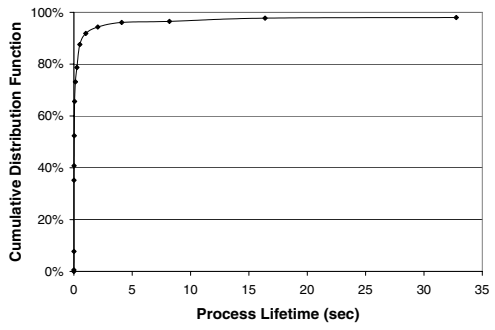


Figure 5: Cumulative distribution function of process life times.

5 Evaluation Results

Thus far, we have described how PAVM has been designed, implemented, and later evolved to improve energy efficiency of memory. In this section, we compare the effectiveness of the various power-management techniques implemented in PAVM with existing techniques to reduce memory's energy consumption. We first describe our experimental setup and test workloads, and then present extensive experimental results.

5.1 Experimental Workloads

Our goal in this section is to evaluate the effectiveness of PAVM when running real-world applications in a working system, and see how it compares to some other power-management policies. All workloads are executed on a Pentium 4 PC, with 512 MB of RDRAM (16 devices), running Linux 2.4.18 kernel. We define three types of workloads: Light, Poweruser, and Multimedia. These workloads are composed of different sets of user applications as shown in Table 5. The applications are not meant to be comprehensive, but rather found to be representative of the type of workloads used most often by us and our colleagues on mobile platforms.

Our representative Light workload consists of web browsing, with some e-mailing and word processing, while listening to mp3 music in the background, all run in a windowed, graphical environment. This type of workload is most commonly used on mobile platforms, and the workload is characterized as mostly idle. However, some users (Powerusers), due to the nature of their work (e.g., graphics designing, programming), utilize and stress their systems more. Their workloads can be characterized by repeated periods of low system utilization (designing, coding) followed by periods of high system utilization (rendering, compiling). To simulate this type of workload, we add periodic Linux kernel compilations to generate periods of heavy load on top of the Light workload. With a growing number of multimedia-rich applications, users may impose even heavier workloads on their mobile systems (e.g., 3D gaming, playing video). Multimedia workloads keep the system in a high utilization state continuously for a long period of time. To simulate this and keep workloads consistent across different experiments, we play an MPEG4-encoded movie using the Xine video player in full-screen mode.

T_E	Total elapsed time in the system
T_I	Total elapsed idle time in the system
$t^S_{i,j}$	Total time that node *j* operates in Standby mode while process *i* is active
$t^N_{i,j}$	Total time that node *j* operates in Nap mode while process *i* is active
f	Activity factor of memory transactions[2]
U	Set of all processes in the system
P_A	Power dissipation of a node in Read/Write mode
P_S	Power dissipation of a node in Standby mode
P_N	Power dissipation of a node in Nap mode
V	Set of all nodes in the system

Table 6: Logged data and static parameters for calculating energy consumption.

5.2 Evaluation Methodology

Our PAVM implementation is currently fully operational, and has all the means to communicate with the RDRAM memory controller (i82850 chipset) on our Pentium 4 testbed to manage power by controlling the power state of individual nodes. However, due to a hardware bug found in the chipset [16], the system will hang when instructed to put a node in Nap mode. As a result, this prevents us from directly measuring the actual energy saved, e.g., with a digital power meter. However, by logging detailed information about the state of processes and the state of the system, combined with the information from memory device's datasheet, we can calculate fairly accurately how much energy would be consumed.

Specifically, to accurately calculate energy consumption, we need to log the operating times and memory use characteristics shown in the top portion of Table 6 from the running system. We also need some static system/memory parameters, shown in the bottom portion of Table 6, to complete the energy calculation.

Using these parameters, we can compute the energy consumed with the following equation:

$$Energy = |V|T_I P_i + \sum_{i \in U} \sum_{j \in V} \left(t^S_{i,j} P_S + t^N_{i,j} P_N\right) + f(T_E - T_I)(P_A - P_S). \quad (1)$$

where $P_i = P_N$ or P_S, depending on the power-management scheme used (see next section). This equation consists of three terms. The first is the energy consumed by the memory while the system is idle, and is simply the product of the number of nodes, total idle time, and either P_N or P_S, depending on whether the nodes are kept in Nap or Standby modes when system is idle. The second term computes the energy for keeping nodes in Nap and Standby modes while the system is not idle. This is a double summation over all processes and all nodes, where we weight the total time a particular process keeps a particular node in Nap and Standby modes by P_N and P_S, respectively. The last term reflects the additional energy required to actually read/write data from/to a memory device in Standby mode, and is a product of the total non-idle time, the additional power dissipated in Read/Write mode over Standby, and an activity factor, f, that gives the total number of memory transactions as a fraction of the maximum number possible when a device is kept in Read/Write mode (i.e., peak memory bandwidth).

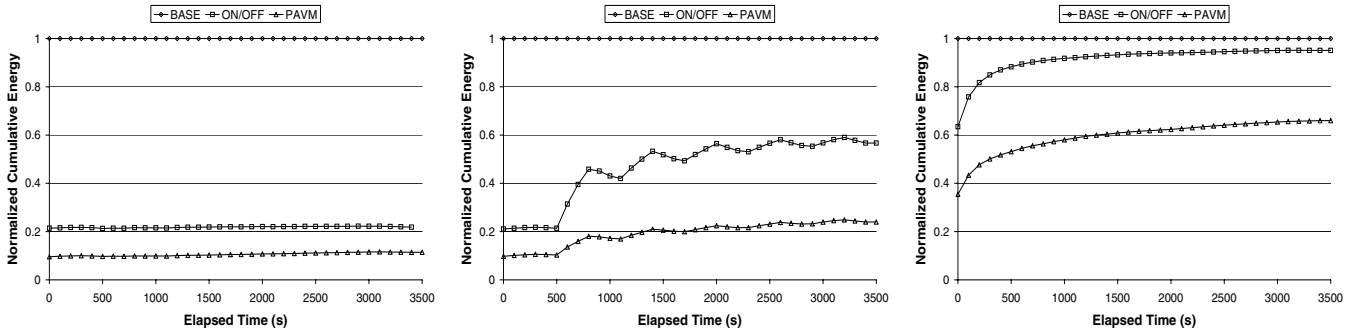
[2]The activity factor, f, is obtained by dividing the number of memory transactions by the maximum possible number of memory transactions during non-idle time, $T_E - T_I$. The dividend is obtained from performance monitoring registers available on most modern processors, and the divisor is derived from the memory device's datasheet.

5.3 Comparison of Basic Techniques

In this section, we compare three basic memory power-management techniques: the default built-in power-management policy implemented in current RDRAM memory controllers (*Base*), a simple *On/Off* technique, and our initial *PAVM* implementation. Recall from Section 2 that, under the *Base* policy, the controller keeps devices in Standby mode, and quickly switches them to Attention mode when accessed. The *On/Off* technique simply involves putting all nodes into Nap mode upon detecting system is idle, and restoring all nodes to Standby when any process is ready to run. It requires minimal kernel modifications to implement, and is worth considering here for its simplicity. PAVM, as described in Section 4.1, is compared with these two methods.

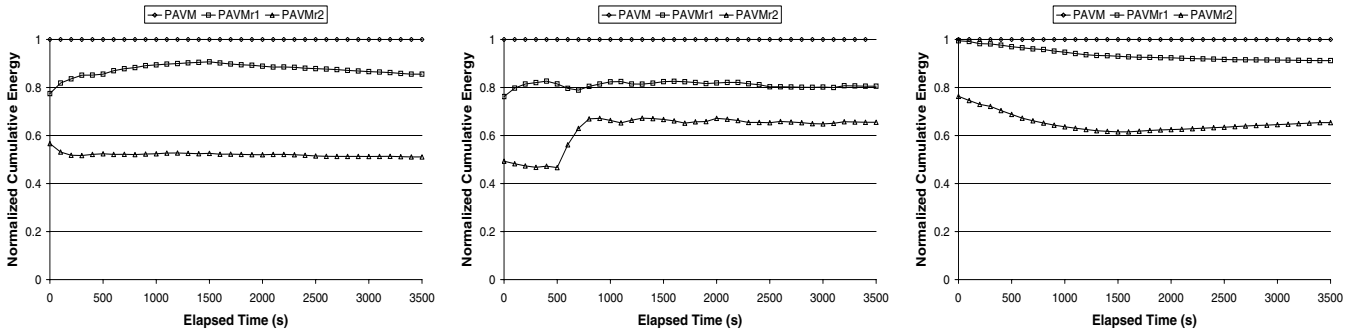
As the *Base* policy always keeps nodes in Standby, while the other two put all nodes into Nap mode when the system is idle, we use $P_i = P_S$ for *Base* and $P_i = P_N$ for the other policies when computing energy with Eq. 1. As neither *Base* nor *On/Off* uses Nap mode while processes are running, the second term in Eq. 1 simplifies to $|V|(T_E - T_I)P_S$ for these two policies.

To show how these power-management policies perform in real systems, we run the three workloads described earlier. The results are shown in Figures 6(a–c) for Light, Poweruser, and Multimedia workloads, respectively. Each graph shows cumulative energy consumed over time, normalized with respect to the *Base* policy. As one can see, for Light and Poweruser workloads, the simple *On/Off* policy performs well since it can exploit the large amount of idle time in the system to put nodes into Nap mode. With the Multimedia workload, idle time in the system is minuscule, and therefore, the *On/Off* policy approaches the *Base* policy at the end of the workload. PAVM, on the other hand, not only exploits idle time, but also reduces memory power dissipation when processes are actively running. Compared to the *On/Off* policy, it can save an additional 48–66%, 51–63%, 30–62% of energy for Light, Poweruser, and Multimedia workloads, respectively.



(a) Light workload (b) Poweruser workload (c) Multimedia workload

Figure 6: Cumulative energy for PAVM and On/Off policies, normalized to that of Base policy when running Light, Poweruser, and Multimedia workloads.



(a) Light workload (b) Poweruser workload (c) Multimedia workload

Figure 7: Cumulative energy for PAVM with library aggregation (PAVMr1) and PAVM with both library aggregation and page migration (PAVMr2), normalized to that of the initial PAVM implementation.

	Light	Poweruser	Multimedia
Base	4100 mW	4118 mW	4230 mW
On/Off	892 mW	2324 mW	3991 mW
PAVM	465 mW	986 mW	2687 mW
PAVMr1	397 mW	791 mW	2442 mW
PAVMr2	237 mW	646 mW	1725 mW

Table 7: Average memory power consumption for the different power-management policies, running various workloads over a one hour period.

5.4 Comparison of Advanced Techniques

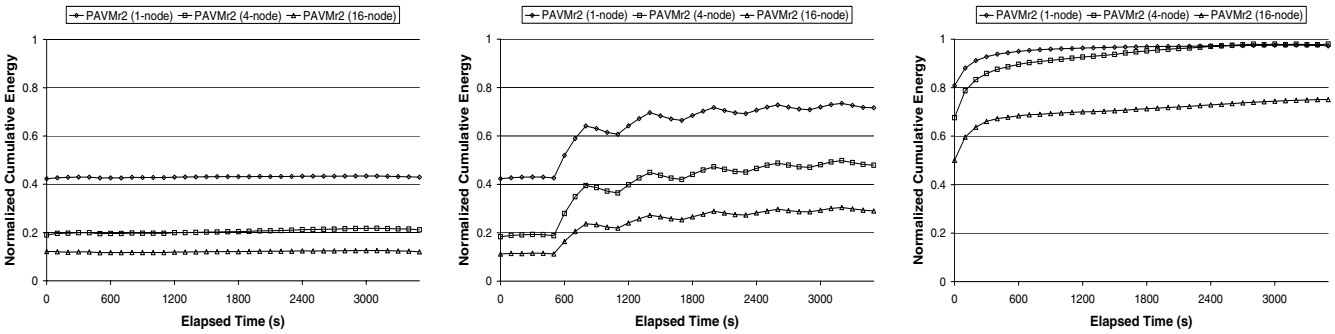
Although the initial implementation of PAVM does very well compared to other basic techniques, we can conserve even more energy using more aggressive policies. In this section, we compare three versions of PAVM: the initial implementation of PAVM, revision 1 that uses library aggregation (PAVMr1), and revision 2 that also includes page migration (PAVMr2). Both of the aggressive policies try to keep nodes in the Nap mode longer, i.e., reducing $\sum t_{i,j}^{S}$ in the second term of Eq. (1), to realize significant additional energy savings.

We repeat the set of workloads under PAVMr1 and PAVMr2 policies, and the resulting energy consumption is plotted in Figures 7(a-c), normalized to that of the initial PAVM implementation. PAVMr1 saves an additional 0–20% and PAVMr2 saves an additional 25–50% of the energy relative to the initial implementation. It is interesting to note the jump at the 10-minute mark in the Poweruser workload for PAVMr2. This is the point at which the periodic Linux kernel compilation first runs. Since kernel compilation creates many short-lived processes that start and complete between invocations of *kmigrated*, page migration does not help these processes, although it continues to be effective for the long-lived ones. Therefore, the benefit of page migration diminishes if short-lived processes dominate in the system.

The absolute average power dissipated for all of the power management techniques is summarized in Table 7. The *Base* system tends to draw close to a constant amount of power, since all nodes stay in Standby mode, with some small increases corresponding to the greater number of memory transactions in the Poweruser and Multimedia workloads. The energy savings realized vary greatly with the workload, and up to 94% reduction is seen with a lightly-loaded system. However, even with the very heavy Multimedia workload, 59% memory power reduction is realized.

5.5 Page-Migration Overhead

There is a significant energy improvement for PAVMr2 over PAVM, but this comes at a cost of page-migration overheads.



(a) Light workload (b) Poweruser workload (c) Multimedia workload

Figure 8: Effects of DDR's physical memory configuration on power dissipation under PAVMr2 when running Light, Poweruser and Multimedia workloads. Cumulative energy is normalized to the DDR Base policy.

Since we migrate pages only when the system is idle, avoiding interference with active processes, there is no direct performance overhead, and only an additional energy penalty is imposed for each page migrated. However, due to the periodic invocation of *kmigrated* and the check for system idle, there is an implicit limitation of migration to only the longer-lived processes. In Section 4.5, we have discussed other possible solutions to limit any overheads.

By logging page migration traffic, we determined that migration, with only the implicit limitation, accounts for only 2.7%, 0.8% and 0.8% of the total memory traffic for Light, Poweruser, and Multimedia workloads, respectively. If we adjust Figures 7(a-c) assuming maximal overhead reduction, we see no perceivable differences, so explicit attempts to reduce these overheads are not fruitful. Due to the significant energy savings and a fairly low overhead observed, page migration is beneficial in almost all circumstances.

5.6 Other Memory Architectures

We have discussed our power-management techniques primarily in the context of RDRAM architecture, but they are also applicable to other SDRAM architectures that support multiple operating power levels. In Figures 8(a-c), energy consumption is shown when running the same set of workloads discussed above, assuming a 4-node DDR memory configuration using PAVMr2. The cumulative energy is normalized against the default hardware-implemented policy for DDR. We still obtain significant energy savings, but not as much as with the previously-described RDRAM configuration.

There are two reasons for this. First, the power difference is much smaller between the DDR modes that correspond to RDRAM's Standby and Nap modes. Therefore, for DDR, putting nodes in "Nap" mode shows a smaller relative energy savings. Second, and more importantly, the notion of a node is coarser-grained for DDR than RDRAM. As discussed earlier, power management for DDR can only be done at the module-level, whereas in RDRAM, power can be adjusted at a device-level granularity, resulting in a much larger number of nodes.

To show the effect that the number of nodes in the system has on energy savings, in Figure 8, we also compare energy consumption assuming 1-node and 16-node DDR configurations. For a 1-node configuration, PAVM basically degenerates to the *On/Off* policy, since the only node must be active for all processes. As the number of nodes increases and the size of each node decreases, PAVM has finer-grained power management control, and yields greater energy savings. Once the number of nodes is increased beyond a certain point, we would expect decreasing, and possibly negative, marginal returns due to operational overheads of managing a large number of nodes. Finding the sweet spot that provides the maximum energy savings is system-/memory- dependent and beyond the scope of this paper. However, we believe that the 8- to 16-node granularity provided in most RDRAM configurations is not far from this sweet spot for typical mobile workloads. Furthermore, assuming that 4 nodes are available in a DDR system is probably optimistic, since in real systems, we are more likely to see 1-node and 2-node configurations, especially on mobile platforms. The results for SDR is similar to DDR, and due to the space limitation, are not shown here.

6 Related Work

Conserving energy in mobile and embedded systems is becoming an active area of research as hardware components are becoming more power-hungry than ever, and as battery technology is not able keep up with the growing demands. By exploiting the ability of modern hardware components to operate at multiple power levels, recent research has demonstrated that a significant amount of energy can be conserved. Due to the high-peak power demands of the processor, a large body of work has focused on reducing processor energy consumption. Weiser *et al.* [38] first demonstrated the effectiveness of using Dynamic Voltage Scaling (DVS) to reduce power dissipation in processors. Later work [2, 11, 14, 15, 25, 29–32] further explored the effectiveness of DVS techniques in both real-time and general-purpose systems.

There is also a large body of work that focused on reducing

power in other system components, including wireless communication [10, 18, 21, 36], disk drives [6, 7, 22, 24], flash [5, 28], cache [1, 19, 37], and main memory [3, 4, 8, 9, 23], while others [12, 26, 35, 40] explored system-level approaches to extend/target the battery lifetime of systems, as opposed to saving energy for individual components.

Among the works dealing with main memory energy, in [8, 23], Lebeck *et al.* studied the effects of various static and dynamic memory-controller policies to reduce power dissipated by the memory using extensive simulations. However, they assumed having additional hardware support to do very fine-grained idle time detection for each device so the controller can correlate this idle time with a power state for each device. In a later work, they used a stochastic Petri Nets approach to explore more complex policies [9]. Our work differs significantly in not assuming any additional hardware support or a particular memory architecture. Moreover, by elevating the decision-making to the OS level, we can use information known to the OS to conserve more energy without degrading performance. Finally, we have fully implemented a power-aware VM system that handles the complexities of a real, working system, and demonstrated its effectiveness when running real-world applications.

Delaluz *et al.* [3] took a *compiler-directed* approach, where power-state transition instructions are automatically inserted into compiled code based on offline profiling. The major drawback of this approach is that the compiler only works with one program at a time and has no information about other processes that may be present at runtime. Therefore, it needs to be either less aggressive or else it can trigger large performance and energy overheads when used in a multitasking system. This approach, however, is appropriate for DSP-like platforms where single-application systems are common.

Delaluz *et al.* [4] later showed a simple scheduler-based power-management policy. The basic idea is similar to our work, but is of much more limited scope. In our work, much effort is put into making the underlying physical page allocator to allocate pages by collaborating with the VM through a NUMA management layer so the energy footprint is reduced for each process, whereas they rely on the default page allocation and VM behaviors. As we have seen in Section 4.2, a substantial amount of power-saving opportunities remain unexploited even with our rudimentary implementation of PAVM, let alone when randomly allocating pages using the default page allocator. In [23], it was also noted that the default page allocation behavior has a detrimental impact on the energy footprints of processes. Second, we have explored advanced techniques such as library aggregation and page migration which are necessary for reducing memory footprints when complex sharing between processes in real operating systems is involved. Finally, in their work, the active nodes are determined using page faults and repeated scans of process page tables. Although this ensures only the truly active nodes are detected, it is intrusive and involves high operational overheads. In contrast, we take every precaution to avoid performance overheads and hide any unavoidable latencies in our implementation, and the end result is a PAVM system that can save a significant amount of energy with only a very small performance overhead.

7 Discussion

In the current implementation, there are two limitations that we do not fully address. First, we do not consider direct memory access (DMA) by other hardware components on nodes that may be in reduced power states, which may result in performance degradation. This can be mitigated by ensuring that DMA uses only pages within a pre-defined physical memory range (e.g., the first node), which, due to the use of library aggregation, is almost always in Standby mode.[3]

Second, kernel threads that run in the background may touch random pages belonging to any process in the system. Since these maintenance threads are invoked fairly infrequently, a simple solution is to treat these as special processes and turn on all nodes when they are invoked to avoid performance degradation.

8 Conclusion and Future Work

Due to better processing technology and a highly competitive market, systems are equipped with bigger-capacity and higher-performance main memory as workloads are becoming more data-centric. As a result, power dissipated by the memory is becoming increasingly significant. In this paper, we have presented the design and analysis of power-aware virtual memory (PAVM) to reduce total memory energy expenditure by managing power states of individual memory nodes. We have also shown a working implementation of PAVM in the Linux kernel, and described how it was later evolved to handle complex memory sharing among multiple processes and between processes and the kernel in a modern operating system.

By performing extensive experiments with real applications, we are able to show that even with a rudimentary version of PAVM, we can save 34–89% of the energy normally consumed in a 16-device RDRAM memory configuration. By applying more advanced techniques such as DLL aggregation and page migration in PAVM, we are able to reduce energy dissipation by an additional 25–50%. We have also shown the applicability of this approach for other SDRAM architectures such as DDR and SDR, which can also benefit greatly under PAVM.

We have used a NUMA abstraction to organize and manage memory in our PAVM implementation, and have borrowed some NUMA concepts such as the notion of a node

[3] Note that due to limitations in older ISA hardware, Linux for x86 already has support to limit DMA transactions to the first 16 MB of memory (i.e., within the first node).

and the page-migration technique. In the future, we would like to explore if other NUMA techniques, such as page replication, can be effective in the context of energy conservation. In addition, we would also like to investigate the interactions between OS-controlled and hardware-implemented power-management policies to further decrease energy consumption of memory.

References

[1] R. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *International Symposium on Low Power Electronic Design*, 1998.

[2] T. D. Burd and R. W. Brodersen. Energy efficient CMOS microprocessor design. In Trevor N. Mudge and Bruce D. Shriver, editors, *Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture*, pages 288–297. IEEE Computer Society Press, 1995.

[3] V. Delaluz and *et al*. Dram energy management using software and hardware directed power mode control. In *International Symposium on High-Performance Computer Architecture*, 2001.

[4] V. Delaluz and *et al*. Scheduler-based dram energy power management. In *Design Automation Conference 39*, 2002.

[5] Fred Douglis, Ramon Caceres, M. Frans Kaashoek, Kai Li, Brian Marsh, and Joshua A. Tauber. Storage alternatives for mobile computers. In *Operating Systems Design and Implementation*, pages 25–37, 1994.

[6] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. In *USENIX Winter*, pages 292–306, 1994.

[7] Fred Douglis, Padmanabhan Krishnan, and Brian Bershad. Adaptive disk spin-down policies for mobile computers. In *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Computing*, 1995.

[8] X. Fan, C. S. Ellis, and A. R. Lebeck. Memory controller policies for dram power management. In *International Symposium on Low Power Electronics and Design*, 2001.

[9] X. Fan, C. S. Ellis, and A. R. Lebeck. Modeling of dram power control policies using deterministic and stochastic petri nets. In *Workshop on Power-Aware Computer Systems*, 2002.

[10] Laura Marie Feeney and Martin Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *IEEE INFOCOM*, 2001.

[11] Krisztian Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM'01*, 2001.

[12] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles*, pages 48–63, 1999.

[13] Richard A. Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. In *USENIX Winter*, pages 201–212, 1995.

[14] K. Govil, E. Chan, and H. Wassermann. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st Conference on Mobile Computing and Networking MOBICOM'95*, 1995.

[15] Flavius Gruian. Hard real-time scheduling for low energy using stochastic data and DVS processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'01*, 2001.

[16] Intel. http://www.intel.com/design/chipsets/specupdt/.

[17] Intel. http://www.intel.com/design/mobile/perfbref/250725.htm.

[18] Christine E. Jones, Krishna M. Sivalingam, Prathima Agrawal, and Jyh-Cheng Chen. A survey of energy efficient network protocols for wireless networks. *Wireless Networks*, 7(4):343–358, 2001.

[19] M. Kamble and K. Ghose. Energy-efficiency of vlsi caches: A comparative study. In *Proc. of International Conference on VLSI Design*, 1997.

[20] Donald E. Knuth. The art of computer programming. volume 1, pages 435–455, 1968.

[21] Robin Kravets and P. Krishnan. Power management techniques for mobile communications. In *Proceedings of the 4th Conference on Mobile Computing and Networking MOBICOM'98*, 1998.

[22] P. Krishnan, P. Long, and J. Vitter. Adaptive disk spin-down via optimal rent-to-buy in probabilistic environments. In *Proc. of International Conference on Machine Learning*, pages 322–330, 1995.

[23] Alvin R. Lebeck and *et al*. Power aware page allocation. In *Architectural Support for Programming Languages and Operating Systems*, pages 105–116, 2000.

[24] Kester Li, Roger Kumpf, Paul Horton, and Thomas E. Anderson. A quantitative analysis of disk drive power management in portable computers. In *USENIX Winter*, pages 279–291, 1994.

[25] Jacob Lorch and Alan J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 50–61, 2001.

[26] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Operating-system directed power reduction. In *International Symposium on Low Power Electronics and Design*, pages 37–42, 2000.

[27] Yung-Hsiang Lu, Luca Benini, and Giovanni De Michelli. Power-aware operating systems for interactive systems, 2002.

[28] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *To appare in Proceedings of the 27th Hawaii Conference on Systems Science*, 1994.

[29] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low-Power*, 2000.

[30] Trevor Pering, Tom Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'00*, 2000.

[31] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *18th ACM Symposium on Operating Systems Principles*, 2001.

[32] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM'01*, 2001.

[33] Rambus. http://www.rambus.com/technology/quickfind_documents.shtml#datasheets.

[34] Rik V. Riel. http://www.surreal.com.

[35] Tajana Simunic, Luca Benini, Peter Glynn, and Giovanni De Micheli. Dynamic power management for portable systems. In *International Conference on Mobile Computing and Networking*, pages 11–19, 2000.

[36] M. Stemm and R. H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Communications, vol.E80-B, no.8, p. 1125-31*, E80-B(8):1125–31, 1997.

[37] C. Su and A. Despain. Cache design tradeoffs for power and performance optimization: A case study. In *Proc. of the International Symposium on Low Power Design*, pages 63–68, 1995.

[38] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23, 1994.

[39] W. Wulf and Sally McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[40] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *International Conference on Archtectural Support for Programming Languages and Operating Systems*, 2002.