An Implementation of Page Allocation Shaping for Energy Efficiency

Matthew E. Tolentino, Joseph Turner, and Kirk W. Cameron

Department of Computer Science
Virginia Polytechnic Institute and State University
2200 Kraft Drive, Blacksburg, VA, 24060
{metolent, turnerj9, cameron}@cs.vt.edu

Abstract

Main memory in many tera-scale systems requires tens of kilowatts of power. The resulting energy consumption increases system cost and the heat produced reduces reliability. Emergent memory technologies will provide systems the ability to dynamically turn-on (online) and turn-off (offline) memory devices at runtime. This technology, coupled with slack in memory demand, offers the potential for significant energy savings in clusters of servers. However, to realize these energy savings, OS-level memory allocation and management techniques must be modified to minimize the number of active memory devices while satisfying application demands. We propose several page shaping techniques and structural enhancements to proactively and reactively direct allocations to a minimal number of devices. To evaluate these techniques on real systems, we implemented these shaping techniques in the Linux kernel. Experiments using our OS extensions coupled with a simple history-based heuristic (to track demand and control state transitions) yield up to 60% energy savings with less than 1% performance loss for various benchmarks including lmbench and SPEC.

1. Introduction

Scientific computing platforms are rapidly approaching petascale. Such systems may have thousands or tens of thousands of processors, tens or hundreds of terabytes of memory, and hundreds of petabytes of disk space [1]. The power consumption of petascale systems, therefore, will likely be tens to hundreds of megawatts, requiring specially designed facilities to house, cool, and power these systems. Power and cooling budgets may soon rival the cost of the hardware.

A key design challenge for HPC is to reduce the power consumption of emergent systems while maintaining stringent performance constraints at reasonable cost. Processors typically account for the largest amount of power in a high-performance cluster, yet memory power is significant [3, 16]. For example, the IBM Bluegene at LLNL uses 32 terabytes of main memory that consume approximately 70 kilowatts of peak power. That's a maximum of $1200 per week for memory energy alone excluding cooling costs.

Several methods for reducing the power consumption of the processor have been proposed, such as DVS, DFS, and clock gating, [2]. These techniques all rely on the concept of *slack* in processor utilization for a given workload. When processor demand decreases, during I/O or network traffic for instance, the supply voltage or frequency is decreased to conserve power. Numerous studies have shown that clever scheduling of low power modes during computationally slack periods results in reduced energy consumption with minimal performance loss.

As in processor utilization, slack in system memory demand provides opportunities for energy savings in the memory subsystem through power-mode scheduling. The key to conserving energy in memory is to offline memory devices whenever possible without impacting performance. During slack periods, if we minimize the number of online memory devices we reduce the total energy consumption of the system. However, to avoid degrading performance we must be able to adapt to increasing demand by quickly turning on additional memory. Therefore, by dynamically adapting the amount of online system memory according to workload demand, we can minimize the energy consumption of memory.

Previous mechanisms for decreasing the power consumption of the memory subsystem have been hardware-centric and focused primarily on mobile devices [9, 15]. Other approaches have extended the operating system scheduler to manage power state transitions through per-process memory reference accounting [8]. Huang et al. leveraged NUMA memory management infrastructure to reduce memory energy consumption on a per-process basis [13]. Li et al. proposed control algorithms to reduce the energy consumption without hurting performance in memory hierarchies and disks [17].

Emergent memory technology provide systems with the ability to dynamically turn-on (*onlining*) and completely turn-off (*offlining*) memory devices at runtime

[1] 1-4244-0910-1/07/$20.00 ©2007 IEEE

[11]. Unfortunately, direct application of previous power-aware memory approaches to onlining and offlining memory devices are problematic. First, a device can only be powered off if it contains no allocations. Since many operating systems do not support transparent page migration, this is not typically possible. Second, even with support for directed page migration, the performance-driven allocation policies of the OS may stripe data across devices making offlining impractical since performance penalties will be severe. Third, monitoring memory usage per process to schedule device transitions is not scalable to HPC systems with tens of thousands of processes.

In this paper, we address the problems of extending the operating system to support onlining and offlining memory devices for systems with dense memory topologies. Inspired by network traffic flow research, we propose several OS-level page allocation and management *shaping* techniques to proactively and reactively direct allocations to a minimal number of devices. We also review the structural changes necessary to enable memory to be onlined and offlined at runtime. We extend the Linux operating system to support these shaping techniques and structural enhancements. Using our kernel implementation on *real* systems, we evaluate the performance impact of our modifications against an unmodified kernel. Experiments using a simple history-based heuristic for controlling the power state transitions of memory devices, our techniques yield up to 60% energy savings in memory with less than 1% performance loss.

2. Structural Changes

Collectively, the set of memory devices in a system forms the usable physical address space managed by operating systems. Due to electrical constraints, memory devices (e.g. DIMMs) are usually only added or removed when a system is powered off. So from the OS perspective, the size of the physical address space or aggregate capacity of all memory devices is fixed at boot time. Accordingly, memory related data structures within the kernel have been traditionally designed to manage a fixed memory device set at runtime. To reduce power consumption, we modified these data structures to cope with transient memory devices, enabling devices to be easily onlined or offlined with minimal overhead. This section highlights these changes.

2.1 Traditional Page Frame Accounting

Most operating systems use a frame table to track the state of usable page frames within the physical address space. The frame table is generally organized as a contiguous linear array such as the *cmap* in BSD [19], the *Ram Tab* in Nemesis [12], the *resident page structure* in Mach [20], and the *memory map* in Linux [10]. Because memory capacity is not expected to change at runtime, a statically sized frame table is used that covers the usable physical address space [4]. This simplifies the implementation of frame state lookup logic as the page frame number can be used as an index within the frame table.

2.2 Mapping Page Frame Sets to Devices

To effectively manage the power states of memory devices we need to track and manage page frame allocations by device. Since devices are mapped into the physical address space and frame tables are used to track frame state, we partition the traditional frame table into sets of frame tables, one for each power-manageable, memory device. For example, in a system that has 8GB of system memory with a power-managed memory granularity of 1GB (i.e. memory device size), the system-level frame table would be composed of eight 1GB page frame sets.

By partitioning the frame table into discrete sets, we accomplish several objectives. First, we gain the capability of tracking page utilization of each memory device capable of being power managed; that is, we can easily discern how many frames are currently allocated or free by simply scanning individual frame tables. Second, we do not waste memory on structures for memory devices that are offline. If we offline a memory device, we can easily free the memory consumed by the associated frame table. Since large frame tables can have a significant memory footprint [10], we minimize the spatial overhead of managing offline memory devices by only allocating sufficient space for online memory devices. This maximizes the memory available for applications in any given memory configuration. Third, each frame table may be dynamically sized to account for memory devices of any capacity or even multiple memory devices with interdependent power states.

3. Page Allocation Shaping

To minimize the energy consumption of dense memory topologies, we need to be able to transition memory devices into lower power states. However, devices that satisfy page allocations may not be transitioned into lower power states without incurring significant latencies upon subsequent accesses. Because lower power states cause higher access latencies, the mapping of pages to frames becomes critical to performance.

In this section, we first briefly discuss page frame allocation in several operating systems and identify the challenges involved in transitioning memory devices into low power states. We then propose and compare three approaches to aggregate page allocations to a minimal set of memory devices.

3.1 Current Allocation Policies

Most operating systems maintain several lists to track page frame state as memory demand changes. For example, BSD variants use *active, inactive, cached,* and *free* lists [19], Solaris uses *free and cache* lists [18], and Linux uses *active, inactive,* and *free* lists [10]. Page frames traverse the lists according to their state and reference frequency. Using multiple lists for currently allocated page frames allows for further delineation between allocated types and has been the focus of memory management research for decades [5-7, 12, 14, 20, 21]. As evidenced by the lists used in these operating systems, page frames are fundamentally either *allocated* or *free*; thus for the purposes of our discussion we shall refer to page frames as being in one of these two states.

Allocated page frames are those that are currently in use. These could include frames mapped into the address space of processes as a result of `malloc` allocations, frames used for I/O transfers or to hold file system data, or even those used for device drivers or kernel data structures. Once a frame is allocated it is removed from the pool of free frames and placed onto a list that tracks its state. Allocated frames are returned to the free pool once explicitly freed or remain unreferenced for some interval.

Frames are often not immediately moved to the free list based on the prediction the page will be referenced again in the future. For example, the buffer and page caches retain previously referenced pages in memory rather than flushing data and returning frames to the free lists. By retaining pages in memory, future references are satisfied quickly by simply mapping the frame into the address space of the requesting process. Such in-memory caches improve performance for workloads that read or modify pages repeatedly. However, workloads with minimal file system I/O interaction, such as computationally-intensive scientific codes, do not tax these caches. For these workloads, these caches often consume significant memory and do not yield significant performance benefits.

Controlling the allocation of all free page frames in the system is the responsibility of a *frames allocator* [12]. When a page frame allocation request arrives, the frames allocator determines which page frame shall satisfy the request. As page frames are continually allocated and freed by the frames allocator, a new allocation request may be satisfied from any valid region in the physical address space. Since the location of each allocation is based on the dynamic memory allocation characteristics of all applications executing on the system preceding the arrival of the request, two back-to-back requests may be mapped to different memory devices.

This behavior is evidenced by the binary buddy allocator used in Linux [10]. The buddy allocator maintains blocks of contiguous page frames by power-of-two size. Several lists are used to aggregate blocks of increasingly larger contiguous page frames. When an allocation request arrives, the request size determines which lists will be searched to satisfy the request. If the list with the optimal order is empty the next list of higher order is searched. Assuming the next list is not empty, a free block (e.g. set of frames) is extracted from the list and split in half. One half is used to satisfy the allocation request and the other half is moved to the next lower order list. As blocks are continuously allocated, partitioned, freed, and moved between lists, contiguous memory regions become fragmented. Since each list is unordered with respect to the address space, allocated frames are selected based solely on request arrival relative to previous allocation and free operations. The effect of this system is that allocated pages are scattered throughout the physical address space. In the worst-case, all memory devices must be retained in a high power state even though only the capacity of a few devices is necessary to satisfy page demand.

Figure 1a illustrates this scattering effect. There are 8 memory devices in this system, each containing 2 pages for a total of 16 pages. Consider an application that allocates a total of 8 pages. As a result of page faults, page frames are allocated individually at regular intervals, resulting in the total allocation of 8 frames by the frames allocator. In the pathological case the memory allocated to the application is scattered across the entire physical address space as depicted by the gray page frames. Because of the distributed allocation pattern, all memory devices must be retained online even though page demand requires only half of the system's capacity.

Given this worst case page frame allocation pattern, we observe several potential solutions: 1) we could migrate pages from frames in sparsely populated memory devices to frames in more densely populated devices. By consolidating frame allocation to a subset of memory devices, unused devices could be transitioned into low power states or even offlined. In our above example, this would reduce the energy consumption of memory by 50% and preserve the existing performance without adding complexity to the frames allocator. However, on real systems we must consider allocation requests that may not be easily migrated, such as those used by device drivers for DMA operations. 2) We could dynamically direct page frame allocation requests to specific regions of physical memory based on the intended use of the page frame. For example, if we knew a set of frames were going to be used for DMA, we could allocate frames from a memory device that we will never try to remove. Similarly, we could direct user-level, dynamically allocated application page frames to regions that are more likely to be removed. 3) We could combine the two approaches and proactively direct page frame allocation to specific regions as well as reactively migrate or swap out currently allocated pages in sparsely allocated devices. The remainder of this section discusses each of these alternatives.

3.2 Reactive Shaping

One approach to the allocation scattering problem is to preserve the allocation characteristics of the frames allocator, but reactively compact allocated page frames into a subset of memory devices. Figure 1 illustrates this approach. Recall the upper half of the figure (1a) shows the worst case frame allocation scheme where pages are scattered throughout devices. Figure 1b shows page placement after migration. Prior to migration all devices were required to remain in a high-power state; however, after

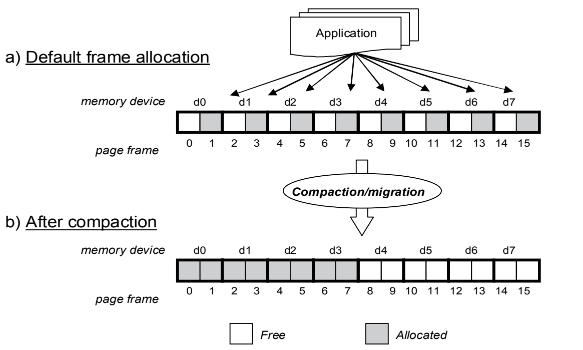

Figure 1. The default allocation policy often results in pages distributed throughout all devices as shown in a). After migration, pages are compacted into a minimal device set enabling devices to be transitioned into lower power states as shown in b). After migration, actual page demand (50% of capacity) is satisfied from the minimal set of memory devices.

migration four devices (half of system capacity) may be offlined or transitioned into a lower power state.

Migrating pages between devices is achieved through several steps. First, a new page frame is allocated. Then the page to be migrated is locked to prevent further access during migration. The page is then copied to the new frame and all references to the old frame are adjusted to point to the new frame. For example, for pages allocated by an application, the page table entry pointing to the old frame is updated to point to the new frame. The page is then unlocked and the old frame is freed.

Randomly moving pages between devices can be costly when devices contain many allocated page frames. For example, consider a system with two devices, each containing 1000 frames. If 900 frames are currently allocated on device 1 and only 50 frames are allocated in device 2, migrating pages from device 1 to device 2 would be suboptimal. To minimize migration costs the pages on device 2 should be migrated to device 1. Consequently, judicious control of page migration is required to minimize overhead.

To avoid this scenario, we scan each memory device to determine how many allocated frames each device contains. We then sort the devices to form two sets. The first set is composed of devices that contain the fewest allocated frames. We migrate pages from these devices to devices in the second set, composed of devices with the most allocated page frames.

Although theoretically, any page can be simply migrated to a different frame on other device, real-system constraints may prevent some pages from being migrated. For example, pages used for DMA operations or performance-centric regions such as those that contain kernel text may incur significant performance penalties to migrate. Considering the first example, frames allocated by a device driver for DMA operations could be freed by temporarily disabling and subsequently restarting the device. However, if the system or application depends on the device for proper operation, such as a storage controller or network interface card, performance could be severely impacted while the device is being reinitialized. Although migration may be possible for all page frames, performance and reliability constraints often limit whether a page may be pragmatically migrated.

In light of these real-world constraints, we further classify pages in terms of their potential for migration. Many operating systems maintain per-page state information that indicates how the page is currently used. We exploit this information to classify pages in terms of those that are pinned *(P)* and others that are easy to move *(E)*. Pages classified as easy-to-move can almost always be moved on demand while pinned pages may never be movable. Generally, pinned pages reduce the opportunity to minimize the number of online memory devices. These classifications also affect our groupings of memory devices as devices that contain pages that may not be moved will not be targets for page migration.

After determining if the set of allocated pages residing on a memory device can be moved, we migrate sets of pages to other areas of the physical address space that map to other memory devices. In essence, we dynamically compact page utilization to a subset of the total number of devices when the number of allocated page frames is less than the total number of page frames.

Figure 2a shows how this approach works using the same memory device configuration as figure 1. In this example, the frame allocator has allocated page frames across 6 of the 8 devices. Gray frames contain allocated pages and white frames are unused. Allocated pages are further marked as *P* and *E*, for pinned and easy to move respectively. While the frames allocator distributed page frame allocations across several memory devices, two memory devices, *d1* and *d7,* have not been used to satisfy any allocations and can be immediately transitioned into a low power state. However, since only 8 of 16 frames are currently allocated, we could optimally turn off 4 of the eight memory devices. We use migration to move the page at frame 7 (an *E* page) to frame 11 as shown by the solid-line arrow, enabling us to turn off device d3. Similarly, we migrate the page at frame 9 to frame 1 (a *P* page) also shown by the solid-line arrow enabling us to turn off device d4. Although we could have migrated our two example pages to any of the available free frames we attempt to move them to devices that have similar allocations. This increases the chance of removing the device later. However, because our approach only moves pages in a reactive manner and does not change how frames are allocated within the physical address space, collocating pages by type may be reversed at the next allocation. For example, if after migrating the page in frame 7 to frame 11, frame 6 is allocated to an *E* page, the page in frame 10 is freed and then populated with a *P* page, then collocating the *E* page in frame 7 wouldn't have been productive.

Instead of migrating pages, we could also free frames by paging pages to disk. As shown in figure 2a rather than migrating the page at frame 7 we could have paged it out to disk. We plan to explore that alternative in future work.

3.3 Proactive Shaping

An alternative approach to page migration is to proactively direct the allocation of frames from specific devices based on the characteristics of the occupying page. To direct allocations, we modify the frames allocator to manage frames in pools according to page type. As before, we differentiate between *P* and *E* pages, and loosely divide the online device set into two sets; one for *E* pages and one for *P* pages. We also add flags to the allocation call interface to distinguish between the page types. By requiring the requester to specify the page type, the frames allocator can direct the allocation to specific devices. For example, when an allocation request for a *P* page arrives, the frames allocator will allocate a frame from the *P* device set. Similarly, when a request for an *E* page arrives, a frame from the *E* device set will be selected.

An example using proactive shaping is shown in figure 3. We divide the available set of frames into two sets, one for *P* pages and one for *E* pages. As in previous examples, page demand is 50% of capacity, but we determine that only two pages are classified as *P* pages, while the remaining 6 allocation requests are for *E* pages. We aggregate the allocated pages into the two frame sets when they are allocated, such that only the minimal device set is consumed by all allocation requests. As a result, half of the memory capacity in the system may be transitioned into lower power states. Since we performed the delineation at allocation time, page migration is not required.

Limitations. Although proactive shaping avoids the overhead of migration, it does have limitations. For example, when unused devices are transitioned into lower power states (such as offline), the number of frames for each type of allocation is reduced. Since we segregate P pages from E pages and offlining devices creates artificial memory limitations, subsequent allocation requests for P pages may be satisfied from the E device set. This condition can lead to fragmentation similar to that originally depicted in figure 1a.

Figure 2b shows how this effect manifests. Unlike figure 2a, we see the *P* pages and *E* pages are aggregated similar to figure 3. However, we also see that device *d2* contains a page that would be better placed on device *d1*; similarly, we observe that there is only a single *E* page on devices *d3* and *d6*. If all these pages were migrated onto common devices, two additional devices could be transitioned into lower power states.

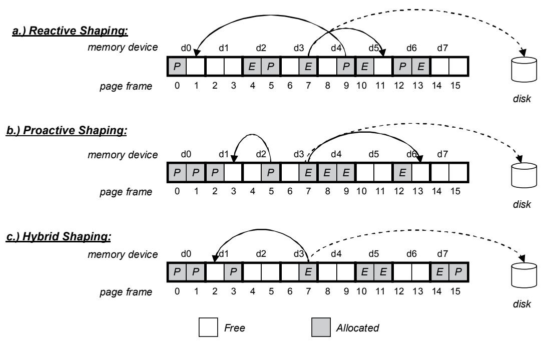

Figure 2. Comparison of three shaping approaches. Reactive shaping uses page migration to aggregate pages onto the minimal device set. Proactive shaping avoids migration costs by placing pages on the minimal number of devices at allocation time. Hybrid shaping combines allocation time placement with page migration to aggregate pages onto the minimal device set.

Implementation Details. For historical reasons, physical memory is coarsely grouped by zone in Linux [10]. However, many supporting architectures use only a subset of the available zones. Consequently, for this discussion we consider an architecture that primarily uses a single zone. Each zone uses a buddy system as the frame allocator to manage free page frames. To direct page frame allocation requests to specific memory devices by allocation type, we use two buddy systems: one for backing E pages and one for P pages. The free area list from which a frame is allocated is determined by checking a flag bit passed into the allocation request. Because this requires only one additional bit-wise comparison and branch instruction, the overhead is trivial. We incorporated this allocation-time direction within the interface functions for allocating and freeing sets of page frames. All other aspects of the buddy allocation algorithm remain unchanged.

Even though this approach minimizes the probability pinned pages will prevent memory removal, it does introduce the possibility of a balancing problem between allocation types. For example, if the number of free frames of either type becomes scarce, this could cause allocation failures for the requested type. To prevent this scenario, we allow for large contiguous areas to be transitioned from one buddy system to the depleted buddy system. If a page frame set is transitioned from buddy system for the *E* frame set to the *P* frame set, the capability of turning off the memory device may be compromised due to pinned pages. However, transferring frame sets between the two systems prevents artificial memory shortages solely because of the delineation between memory request types. A side effect of this approach is a lower bound on the amount of memory that may be de-allocated. However, immovable kernel pages account for a small amount

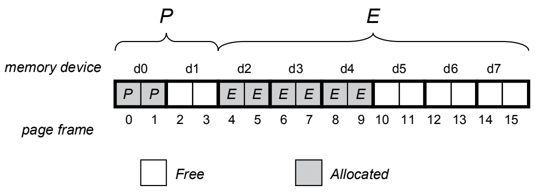

Figure 3. Proactive shaping aggregates pages onto a minimal number of devices at allocation time, avoiding migration overhead. Using proactive shaping, actual page demand (50% of capacity) is satisfied from a minimal set of memory devices.

of total physical memory and at least one memory device must remain powered on to maintain reasonable performance on any static or dynamic memory system.

3.4 Hybrid Shaping

To maximize energy efficiency we propose a third approach called hybrid shaping. Hybrid shaping combines the allocation-time page placement of proactive shaping with the migration capability of reactive shaping. Since proactive shaping directs allocations to devices with minimal overhead, the need for migration is minimized. However, as previously discussed, relying solely on proactive shaping can result in suboptimal allocations across devices over time. Hybrid shaping uses reactive shaping to avoid these inefficiencies by periodically aggregating pages onto a minimal device set.

Figure 2c shows how hybrid shaping work relative proactive and reactive shaping. In this example, page demand is again 50% of system capacity (8 pages, 16 frames). We observe that allocation time placement has aggregated pages by type onto common devices with the exception of device *d7*. Pragmatically, only the *P* page in device *d1* and the *E* page resident in device *d3* are candidates for migration. Because *E* pages are by definition easier to migrate than *P* pages, the page in frame 7 is moved to frame 2 on device *d1*. Optionally, the page in frame 7 may also be paged out to disk; however, this may incur a performance penalty if the page is in active use. After migration half of system capacity may be transitioned into a lower power state.

4. Experimental Results

We implemented all of the extensions discussed in the previous section in a 2.6 x86-64 version of the Linux kernel on a recent Intel 64-bit SMP Xeon processor system with various amounts of memory. For some of the experiments our system contained 3GB. In later experiments we populated the system with 8GB of memory. To evaluate our approach, we modified our operating system extensions slightly to emulate memory topologies by partitioning the physical address space into logical devices. In this way, we can experiment using memory devices of any granularity, within the limits of actual system memory capacity. Using emulation, we can evaluate the impact of dynamically adjusting memory device states using the actual, real-time memory demand of various workloads.

Due to space limitations, we only present energy and performance results using our hybrid approach. Because hybrid shaping is a combination of proactive and reactive shaping techniques, the proactive page placement is integrated into the page allocation process. However, in our implementation page migration is only triggered when a device is offlined by a system-level controller. So, we implemented a root-privileged, application-level daemon using a simple history-based heuristic to monitor page demand and control the power state of all memory devices.

4.1 lmbench Results

For our first evaluation we configured lmbench to run the OS-centric set of benchmarks, including the memory-intensive bandwidth codes, to stress our page allocation and migration modifications. Each run was configured to execute the same codes with the same amount of memory. Additionally, we configured lmbench to use a subset of total memory in the system. We ensure the page demand of lmbench is in-core to evaluate page migration.

Our controller uses observed page demand to predict the number of online memory devices required. Specifically, we retain a configurable number of prior observations (10-20) and use those to predict when to offline devices. To ensure a sufficient number of devices were available to satisfy sudden increases in demand, we retained additional devices online beyond the optimal number of devices. Consequently, the control application aggressively on-lines additional devices when demand increases and off-lines devices only when the number of online devices is greater than any of the observations retained in the recent history buffer.

Figure 4 plots memory demand and online memory as directed by the controller. For this experiment, we first started the controller, after which we started lmbench. At the beginning of the run all memory devices are online and available. After starting the controller, devices are incrementally off-lined to conserve energy due to low memory demand. When lmbench starts there is a spike in memory demand which causes the controller to online additional memory devices quickly, preventing paging. During the initial execution phase, memory demand remains near constant, but then oscillates towards the end. However, because we are using a simplistic history-based heuristic to control power state transitions, some efficiency is lost as noted by the gap between online memory and actual memory demand. We plan to investigate alternative control policies in future work. Despite this efficiency gap, retaining additional memory online preserves performance and still reduces energy consumption. For this experiment, using the hybrid shaping, the controller achieved 56.26% energy savings within the memory subsystem, largely attributable to the limited memory demand of lmbench.

We also ran lmbench against an unmodified base kernel to characterize the performance implications of our changes. For nearly all of the specific codes, there was no discernable difference. The only observable differences were in the memory bandwidth codes. Figure 5 compares the bandwidth results of the memory intensive lmbench codes of our kernel modified to include hybrid shaping and a base kernel. There was less than a 1% difference as a result of our changes.

4.2 SPEC CPU2000 Results

We also experimented using the SPEC CPU2000 benchmarks. In this case, our system was populated with 8GB of memory. Figure 6 shows how memory devices are dynamically scaled to meet memory demand of the SPEC benchmarks using our history-based heuristic. Initially, the full 8GB memory capacity was online and available for use; however, because the SPEC benchmarks do not require significant memory, the controller offlined most of the unnecessary memory devices as also shown in figure 4. Thus, we have omitted the first 5 minutes of the trace in figure 6.

As shown, systemic memory demand increases when SPEC is started and additional devices are onlined to meet the demand. The memory demand of the benchmarks oscillates slightly while executing but never exceeds about 400MB. In this case, a total of about 512MB is more than sufficient to satisfy the demand of all the SPEC benchmarks while executing. Once the benchmarks complete executing, the number of online memory devices is further minimized. This resulted in 81.25% energy savings within the memory subsystem. Our energy savings are significant using SPEC codes due to the low memory demand relative to system capacity. As before, we also ran the SPEC benchmarks on an unmodified base kernel to characterize the performance implications of our changes. Figure 7 shows the performance of several SPEC codes using our kernel with hybrid shaping normalized to the base kernel. The performance loss for these codes is maximally about 1%.

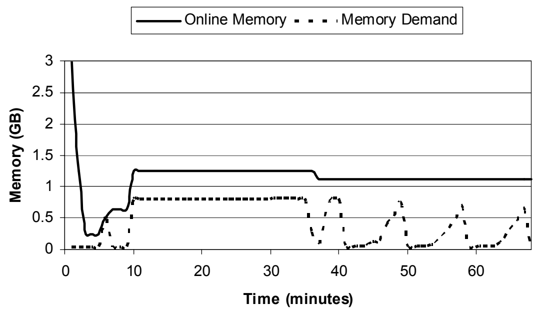

Figure 4. Online memory scaling using hybrid page shaping and simple heuristics while running lmbench. The lower line is the actual memory demand observed and the upper line constitutes the online memory device set which closely tracks memory demand and reduces energy consumption of memory.

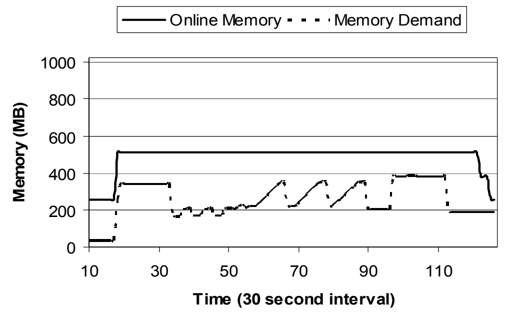

Figure 6. Online memory scaling using hybrid page shaping and simple heuristics while running SPEC CPU2000 benchmarks. The lower line shows actual memory demand and the upper line shows online memory which is adjusted based on memory demand.

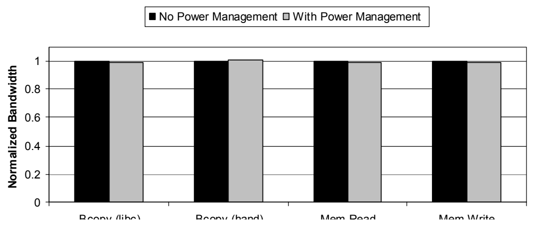

Figure 5. Performance result comparison of lmbench codes with our kernel modified to use hybrid page shaping and an unmodified base kernel. For all codes, there is less than 1% difference.

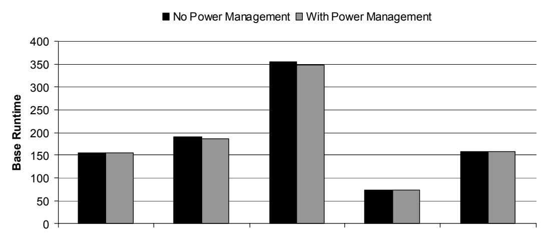

Figure 7. Performance result comparison of SPEC CPU2000 codes with our kernel modified to use hybrid page shaping and an unmodified base kernel. For all codes, there is less than 1% difference.

5. Conclusions

We have quantified the performance and energy for several workloads using a kernel with hybrid page shaping and shown that there are significant opportunities to minimize energy consumption on large systems by taking advantage of variable *slack* in system memory demand. By combining proactive page placement at allocation time and reactive page migration on demand, we achieved significant energy savings (56% for lmbench and 81% for SPEC) with less than 1% performance penalty. Applying these techniques on large memory-dense systems could yield significant cost savings by dynamically scaling online memory based on actual memory demand.

Finally, there is significant work we plan to do on this subject. For example, we want to explore the role of paging to disk in power-aware memory, as it is clear there will be active pages that will not be used for long periods. We also plan to study the tradeoffs for using alternative controllers to monitor page demand and direct the power state transitions of memory devices. Further, we want to study the impact of device off-lining granularity and memory interleaving on controller design. Another study we hope to conduct is implementation of our controllers in hardware.

References

[1] D. H. Bailey, "Performance of Future High-end Computers," in *DOE Mission Computing Conference*, 2003.

[2] L. Benini and G. De Micheli, "System-level power optimization: techniques and tools," *ACM TODAES*, vol. 5, pp. 115-192, 1999.

[3] R. Bianchini and R. Rajamony, "Power and Energy Management for Server Systems," *IEEE Computer*, vol. 37, pp. 68-74, 2004.

[4] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple But Effective Techniques for NUMA Memory Management," presented at SOSP-12, 1989.

[5] A. R. Butt, C. Gniady, and Y. C. Hu, "The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms," presented at *SIGMETRICS'05*, Banff, Alberta Canada, 2005.

[6] R. W. Carr and J. L. Hennessey, "WSCLOCK - A Simple and Effective Algorithm for Virtual Memory Management," presented at *SOSP-08*, 1981.

[7] F. J. Corbato, "A Paging Experiment with the Multics System," MIT MAC Report MAC-M-384, Boston May 1968.

[8] 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*, vol. 50, pp. 1154-1173, 2001.

[9] X. Fan, C. S. Ellis, and A. R. Lebeck, "Memory controller policies for DRAM power management," presented at ISPLED, 2001.

[10] M. Gorman, *Understanding the Linux Virtual Memory Manager*. NJ: Prentice Hall, 2004.

[11] J. Haas and P. Vogt, "Fully-buffered DIMM Technology Moves Enterprise Platforms to the Next Level," in *Technology@Intel Magazine*, vol. 3, 2005.

[12] S. Hand, "Self-paging in the Nemesis Operating System," presented at OSDI-3, 1999.

[13] H. Huang, P. Pillai, and K. Shin, "Design and Implementation of Power-aware Virtual Memory," presented at Usenix 2003 ATC, 2003.

[14] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," presented at USENIX ATC, 2005.

[15] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power Aware Page Allocation," presented at ASPLOS, 2002.

[16] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. Keller, "Energy Management for Commercial Servers," *IEEE Computer*, vol. 36, pp. 39-48, 2003.

[17] X. Li, Z. Li, F. Danvid, Y. Zhou, and S. Kumar, "Performance Directed Energy Management for Main Memory and Disks," presented at ASPLOS, 2004.

[18] R. McDougall and J. Mauro, *Solaris Internals*. Menlo Park, CA: Sun Microsystems Press, 2001.

[19] J. Quarterman, A. Silberschatz, and J. Peterson, "4.2BSD and 4.3BSD as Examples of the UNIX System," *Computing Surveys*, vol. 17, pp. 379-418, 1985.

[20] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," presented at ASPLOS '87, 1987.

[21] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic Tracking of Page Miss Ratio Curve for Memory Management," presented at ASPLOS '04, Boston, 2004.