Journal of Circuits, Systems, and Computers
© World Scientific Publishing Company

Exploiting Temporal Locality for Energy Efficient Memory Management

Euiseong Seo
Division of CS, Korea Advanced Institute of Science and Technology, 335 Gwahangno
Yuseong-gu, Daejeon 305-701, Republic of Korea
[†]*ses@calab.kaist.ac.kr*

Donghyouk Lim
Convergence S/W Research Division, Electronics and Telecommunication Research Institute,
138 Gajeongno, Yuseong-gu, Daejeon 305-700, Republic of Korea
[†]*befree@etri.re.kr*

Seungryoul Maeng
Division of CS, Korea Advanced Institute of Science and Technology, 335 Gwahangno
Yuseong-gu, Daejeon 305-701, Republic of Korea
[†]*maeng@calab.kaist.ac.kr*

Joonwon Lee
Division of CS, Korea Advanced Institute of Science and Technology, 335 Gwahangno
Yuseong-gu, Daejeon 305-701, Republic of Korea
[†]*joon@kaist.ac.kr*

Received 8 April 2007
Accepted 24 February 2008
Electronic version of an article published as Journal of Circuits, Systems, and Computers 2008
©World Scientific Publishing Company (http://www.worldscinet.com/jcsc/)

Memory is becoming one of the major power consumers in computing systems. Therefore energy efficient memory management is essential. Modern memory systems employ sleep states for energy saving. To utilize this feature, existing research activities have concentrated on increasing spatial locality to deactivate as many blocks as possible. However they did not count the unexpected activation of memory blocks due to cache eviction of deactivated tasks. In this paper we suggest a software based power state management scheme for memory which exploits temporal locality to relieve the energy loss from the unexpected activation of memory blocks from cache eviction. The suggested scheme, SW-NAP makes a memory block remain deactivated during a certain tick which have no cache miss over the block. The evaluation shows that SW-NAP is 50% better than PAVM which is an existing software scheme and worse than PMU which is another approach based on the specialized hardware by 20%. We also suggest task scheduling policies which increase the effectiveness of SW-NAP and they saved up to 7% additional energy.

Keywords: power-aware computing, memory management, scheduling

1. Introduction

Recently the use of mobile computers have rapidly grown. They inevitably use the battery as their power source. Therefore for extending the operating time the energy efficiency of those systems are becoming a major design concern.

In general there are two major energy consumers in modern computing equipments. One is a processor and the other is a main memory system[1]. And the power consumption by both of them will continuously increase due to the more integration of the circuits in them and the adoption of high performance technologies such as increasing clock frequency and bus bandwidth.

There have been lots of research on energy efficient processor management[234]. However, relatively few research activities were done on the energy efficiency of memory subsystem. The memory size is increasing according to the increase of the application size and its complexity. The power consumption in a memory subsystem is proportional to its size. Therefore the increased memory size causes more power consumption. As a result the importance of the energy efficient memory management is growing quickly.

Most modern memory architectures provide several operating modes or power states for energy management. There have been some researches to utilize these multiple power states to get energy efficiency. The intuitive approach is to change the state of the memory blocks which are expected to be unused for a while into low-power consuming modes. Naturally existing researches concentrated on reducing active memory blocks at a certain time by modification of memory allocation algorithm. Representatively PAVM(Power-Aware Virtual Memory)[5] places all the memory space allocated by a task at the minimal memory blocks by using NUMA(non-unified memory architecture). By this approach memory blocks used by only the currently executed task are required to be activated.

In this approach, however, the activation of the deactivated memory blocks by the data eviction from processor caches are ignored. While a task is being executed, many cache lines are evicted from the processor and the destinations of the evicted data are spread over the entire memory blocks. As a result due to the unexpected activation the efforts to turn off the unused memory blocks lose much of their effectiveness.

Due to the high spatial locality, most of the memory references are absorbed by processor caches. However, when a task is context-switched to run or starts manipulation over new data sets massive cache misses may occur in a short period and therefore many memory access will be made. In this paper we suggest an optimistic memory management scheme that deactivates all the memory blocks after every ticks to resolve the unexpected activation of deactivated memory blocks from cache eviction. This approach is based on the assumption that few memory blocks will be activated at the next time slice by cache misses.

To increase the effectiveness of the suggested scheme, we also propose two task scheduling policies that dynamically adjust the context switching point based on the actual memory accesses made by the currently running task. The first scheduling policy advances the context switching from the base period when cache misses occur after certain threshold. On the contrary the second one prolongs the context switching for a while after the base period when no memory access occurs during the execution. By both of these algorithms, memory accesses during the entire execution time are expected to be reduced because both of them reduces the context switchings immediately after massive memory read requests.

The suggested schemes were implemented on the Linux kernel. To observe the memory access pattern and energy consumption we used Bochs[6] which is an Open Source x86 architecture emulator. The evaluation results were got by running the prototype implementation in Bochs. The results were compared to the existing research results.

The rest of this paper is organized as follows. Section 2 provides background on the energy efficient feature implemented in modern memory technologies and also related work. Section 3 explains the SW-NAP and two scheduling methods. Section 4 describes evaluation results and Finally, Section 5 concludes this research.

2. Background

2.1. *Operation modes of DRAM*

The basic unit of power control in *DDR-RAM(Double Data Rate RAM)* is called *rank* and in *RDRAM(RAM-Bus DRAM)* it is named as *device*. The operating modes and their properties are similar among different types of DRAM technologies. Figure 1 depicts the power consumption and transition time of power modes which are provided by RDRAM.

A rank or a device is usually a chunk of memory which consists of thousands of consecutive physical pages. However, from now on for the generality we will use *block* to point the basic power management unit. The size of a block is dependant on the memory chip and generally a few tens of mega bytes.

Although RDRAM technology provides several different power states, we will consider only attention and nap mode to prevent the overhead from long transition time. In the rest of this paper, activation and deactivation means the attention mode and the nap mode, respectively.

RDRAM is only able to be read or written when it is in *attention* state. Therefore if a block is deactivated the memory controller will automatically activate it when an access over the block occurs. On the contrary the deactivation is done by delivering state transition command to memory controller.

2.2. *Related Work*

Existing researches tried to reduce the activated memory blocks while tasks are running. The intuitive approach of achieving that is raising spatial locality by gathering data which will be used together and put them into the smallest number of memory blocks possible.

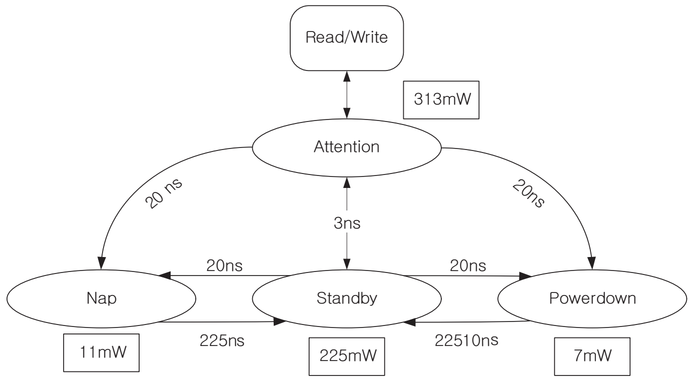


Fig. 1. The state transition diagram of RDRAM

Lebeck et al.[7] proposed a power aware memory allocation scheme based on the first touch sequential algorithm. It allocates physical memory in sequential order. By this, all data used in a task will be placed into a same block or neighboring memory blocks. In a long term, however, much fragmentation will occur and memory used by a task will be sprayed over large number of memory blocks. As a result that phenomena will hinder the spatial locality of allocation. It also assumes dedicated hardware for detecting idle period of memory blocks and deactivating them.

Delaluz et al.[8] proposed scheduler based approach. It keeps a table that tells which memory blocks are used by a certain task. The state of each memory block is transitioned according to the information when a context switch occurs. The memory blocks which are used by the next scheduled task will be activated before executing and naturally the others will be deactivated. This approach works without additional hardware. However, it could not evade from the degradation of the effectiveness by the fragmentation problem again.

Huang et al.[5] suggested power aware virtual memory management scheme, PAVM. It is based on the memory allocation algorithm which shares same philosophy with that of Lebeck et al.. It allocates memory used by a task over small number of memory blocks by gathering them together with utilizing NUMA(Non-Uniform Memory Access) architecture[9]. For reducing the fragmentation, a daemon, *kmigrated*, is designed to move dispersed data into a memory block. Although it works well without hardware support, the unwanted activations from the eviction of processor cache were not considered. If the evicted line is not for the currently executing task, somewhere of the deactivated memory blocks should be activated and it will remain activated until the next context switching.

Lee et al.[10] asserted that also the memory used for kernel buffer should be maintained in the similar manner to PAVM. To achieve that they match each memory block used for kernel buffer to a task which uses the block most frequently and try to pack the kernel buffer block into the memory blocks allocated to the task. The evaluation of PABC(Power-Aware Buffer Cache) which is their prototype implementation showed that the significant portion of total memory blocks was used for kernel buffer and with their approach the energy consumption in those blocks was much reduced. However, they did not mentioned about the unexpected activation from cache evictions in their paper either.

Huang et al.[11] also proposed new hardware-software cooperative approach over PAVM. PMU(Power Management Unit), the dedicated hardware, records the history of the memory references over each memory block. Using that information PMU controls the power state of the memory blocks allocated to a task for each task. If a memory block has not been used for a certain threshold length, then PMU will bring it to sleep state.

In addition to PAVM which improves energy efficiency from unused memory blocks, PMU saves more by identifying unused cycles for each memory block. Naturally PMU approach shows better energy efficiency than the other software-based approach. However, in real world adding a controlling device such as PMU is much harder than adding a software layer. Also the cost for developing as well as manufacturing is a big obstacle to employ the hardware based approaches.

3. Power-Aware Memory Management Scheme

In this section we suggest a software based power-aware memory management scheme,SW-NAP(Software based Napping), which exploits temporal locality. And two task scheduling policies which raise the effectiveness of SW-NAP are also introduced.

3.1. *Memory State Management Policy*

The basic scheduling unit of modern operating systems is a *tick* also known as a quantum or a time slice. In general the length of a tick is from 1 ms. to 10 ms.. When a task is scheduled to run then it will be executed during a few continuously allocated ticks. This is for increasing the hit-rate of TLB(Translation Lookaside Buffer) and processor caches. Therefore generally a task generates small number of cache misses during the execution except the first tick of the allocated ticks for the task.

The activation of a memory block requires about 230 ns.. The power consumption of the activated state is 30 times more than that of the deactivated state. Considering the energy consumption for the state transition, the break-even time of deactivation is 250 ns.[7]. In other words if an idle period of a memory block is expected to be longer than 250 ns. it will be beneficial to deactivate the memory block as soon as possible.

Based on this observation we suggest SW-NAP. It deactivates all of the memory blocks after every ticks excluding the first tick a task was given at scheduling time. SW-NAP utilizes the characteristics that the working set will be activated automatically by themselves. This is different from the traditional approaches that

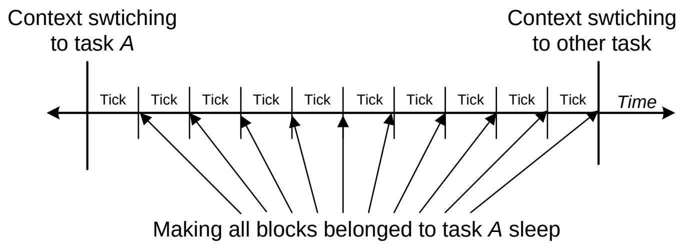

Context swtiching to task *A*
Context swtiching to other task
Tick
Time
Making all blocks belonged to task *A* sleep

Fig. 2. Timeline of SW-NAP

try to find or limit working sets exactly. And SW-NAP exploits temporal locality to extend the deactivated time of memory blocks while others have tried to increase the number of deactivated blocks at a certain time by raising spatial locality. SW-NAP is not exclusive to the spatial locality based approaches. Therefore it is able to be easily combined with those approaches and even more effective with those combinations.

The overhead of SW-NAP are caused by the deactivation of memory blocks and the activation from the cache misses over deactivated memory blocks. With the assumption that there are 32 memory blocks in the target system and the deactivating operation is done for each block in serial order, the deactivation of whole blocks requires about 700 ns.. It is 0.07% of a tick and this is the number for the worst case, when all memory blocks are active. If many blocks would stay deactivated in most of the time, the deactivating time is negligible.

The execution delay from the activation are dependant on the cases. In the worst case when the all 32 blocks are activated in a tick, the activating operations take about 7.5 μsec. This is only 0.7% of a tick. In most cases the size of a block is big enough that the size of working sets are within a few blocks. Therefore few activations are expected to occur in a tick.

3.2. *Power Aware Task Scheduling Policy*

SW-NAP is based on the assumption that the memory access patterns have high temporal locality. However, if a task is switched out shortly after massive cache misses then the accessed memory should be reloaded at the next scheduling of the task because many of the loaded cache lines may be evicted during the execution of the other tasks before the next time slice of the task. In that case the temporal locality will not be fully utilized by SW-NAP. In this section we suggest two scheduling policies to prevent this case and as a result raise the effectiveness of SW-NAP.

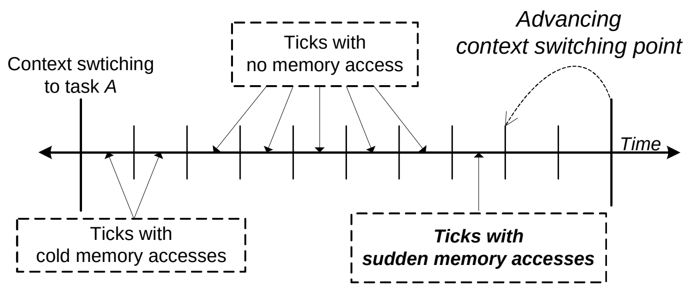

Context swtiching to task *A*
Ticks with no memory access
Advancing context switching point
Time
Ticks with cold memory accesses
Ticks with sudden memory accesses

Fig. 3. The execution flow of ESS

3.2.1. *ESS : Early Context Switching Scheduling*

The memory working set of a task changes as it runs. If a task starts to work on a new working set after a few ticks of working on the former working set, massive cache misses may occur in the next few contiguous ticks. Considering this tendency, ESS switches the current task with the next one immediately if cache misses start to occur near task switching time.

The cache miss or even the number of cache misses occurred can be measured with processor performance measurement counters. They were easily found in most modern processors including *Intel*® *Pentium*^{TM} and *Core*^{TM} architectures and *IBM*® *Power*^{TM} architectures.

ESS is illustrated in Figure 3. Timer interrupt handler counts the number of cache miss occurrences in the last tick. If there is no cache miss, the currently running task is in the execution phase of high temporal locality. It will be executed until it uses up its allocated ticks. On the contrary if there were cache misses in a certain tick before the expiration of the execution, a context switch will be made after the tick. A predefined threshold period in which the cache misses do not affect the scheduling decision should be defined to prevent too early switching. For example if the threshold period is 1/2 of the total given ticks, then if there is cache miss in the front half of the given time, it will not be affected by ESS. Naturally ESS will be applied in the last half of the given time.

ESS may generate overly frequent task switchings especially with the tasks which continually changes its working set. To resolve this ESS should not applied to the tasks that made cache misses in every tick within the threshold period.

3.2.2. *LSS : Lazy Context Switching Scheduling*

The second approach, LSS, is the contrast to ESS. When the last ticks have no cache miss it postpones the context switching and gives more ticks named bonus ticks to the current task after it uses up its given time.

Cache lines which were loaded for a task right before a context switching may be evicted by the next task. In most cases they will be missed again and reloaded

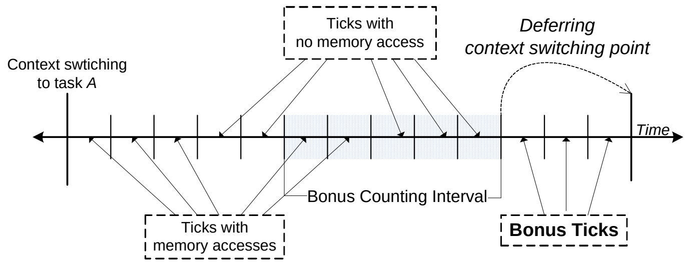


Fig. 4. The execution flow of LSS

in the next scheduled time for that task. By LSS the newly loaded cache lines will stay longer in processor caches and therefore the throughput as well as the cache hit rate will be improved also.

LSS is illustrated in Figure 4. The bonus tick checking routine is executed in the interrupt handler during the bonus counting interval which is generally the last half of given ticks. Naturally the length of the bonus counting interval can be adjusted depending on the characteristics of the tasks. It checks the existence of cache miss during a tick. If there is no cache miss during a certain tick, the task earns one bonus tick. The bonus ticks are used after the task uses up its ticks initially given at the scheduling point. While executing with bonus ticks if the task produces a cache miss then all the remaining bonus ticks will be forfeited and immediately switched out.

In addition to the energy efficiency the cache hit ratio during entire run time will be increased with LSS. However the tasks with few memory accesses or stable working set will get prioritized and this causes unfairness problem. Reducing the number of the initial ticks when the next schedule for the tasks that had benefit from LSS can compensate this unfairness.

4. Performance evaluation

4.1. *Evaluation Environment*

The suggested scheme was implemented on the Linux kernel. However, it was hard to measure the exact memory usage profiles such as ”what memory blocks were activated when” in real systems. Therefore we modified Bochs[6] x86 simulator.

Because the original Bochs has no cache emulation system, cache layer which emulates 8-way set associative cache structure was added. And we also implemented memory controller structure and memory usage profilers. With these additions we can extract the memory usage profiles and measure the power consumption exactly during the actual execution.

We employed PAVM as the memory allocation and management algorithm. Based on it our suggested scheme will manage the operation mode of the memory system during the execution.

Table 1. Experimental Environment

Simulator	Modified Bochs-2.1
Processor Clock	1GHz
Architecture Model	P6 Family
Cache Model	2MB, 64Byte Line 8-Way Set Associative
Main Memory	256MB
Number of Memory Blocks	32
OS Distribution	Fedora Core 3
Kernel version	linux-2.4.20

For evaluation, three applications chosen from SPECCPU2000[12] were used. Actual workloads for the evaluation were running these programs concurrently to simulate multi-tasking environment. The input data set for each programs were unmodified from original benchmark suite.

The evaluation results were compared with PAVM and PMU. For acquiring PAVM kernel, PAVM patch which was released by the authors of PAVM was used. In contrast with PAVM, PMU requires the specialized hardware support. We also implemented PMU hardware support in Bochs.

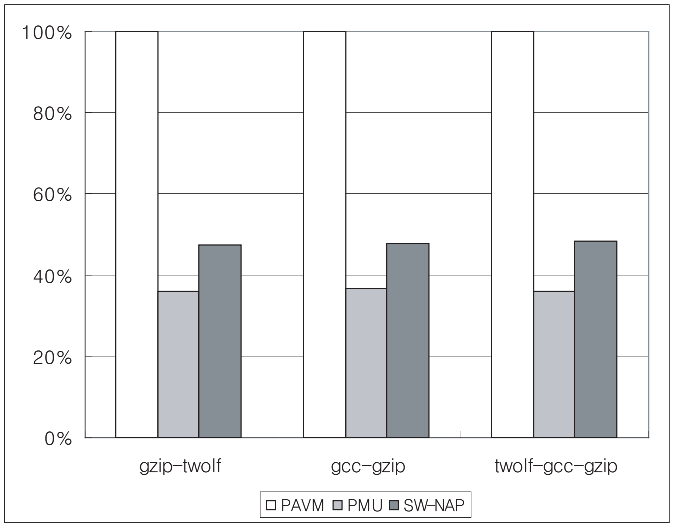
4.2. *Evaluation Results*

The energy consumption and the energy delay product as known as $E \times D$ were measured for the execution of the evaluative tasks.

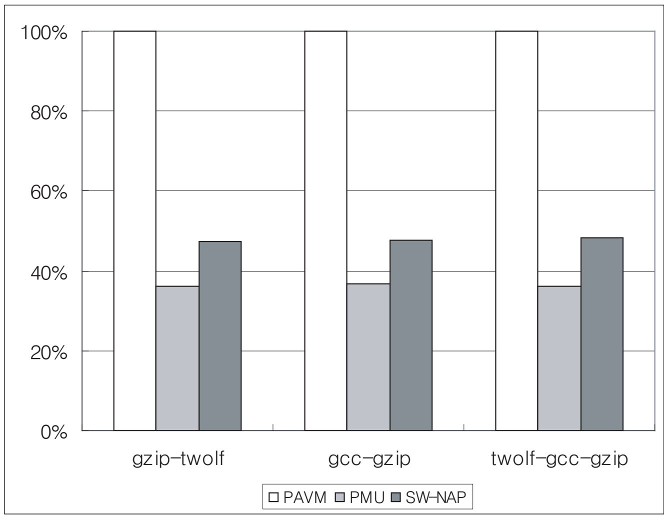
Figure 5 is the normalized energy consumption. As we can see in Figure 5, SW-NAP reduced about 62% energy consumption compared to PAVM while PMU reduced about 65%. PAVM showed the bad result because as aforementioned earlier the intention of PAVM was hindered by the cache evictions. The cache evictions were sometimes scattered over whole memory blocks. But they densely happened within short periods.

As the results showed, the concept that exploits the temporal locality of memory accesses was certainly effective. However, by the aid of dedicated hardware, PMU did it slightly better than SW-NAP with shorter management periods and low overhead to perform operating mode transitions. Generally the differences were under 10% of the PAVM energy consumptions.

The evaluation results for the suggested scheduling algorithms are represented in Figure 6. The decision threshold interval for ESS and the bonus counting interval for LSS was chosen as half of the initially given ticks for a scheduling. Both ESS and LSS showed better energy efficiency than SW-NAP in all cases. When *twolf* was executed LSS showed less energy consumption than ESS. *twolf* requires only 4 Mbytes of memory during the entire execution time[13]. Therefore it has high spatial locality. As a result even in case that there are cache misses at a certain tick for *twolf*, the next tick will tend to have no cache miss at all. In this situation sometimes expedited task switchings by ESS increased the number of cache misses



(a) Energy Consumption



(b) Energy * Delay

Fig. 5. Energy consumption of SW-NAP Normalized to PAVM

rather reduced them. On the contrary LSS was able to utilize the benefit of high spatial locality with extended task switching period. To summarize LSS showed stable energy saving in all cases and ESS worked better only in case that all the tasks have sparse and dynamically changing memory usage patterns.

As well as the energy consumption we measured the total execution time for identifying the overhead of the suggested scheduling policies. As shown in Figure

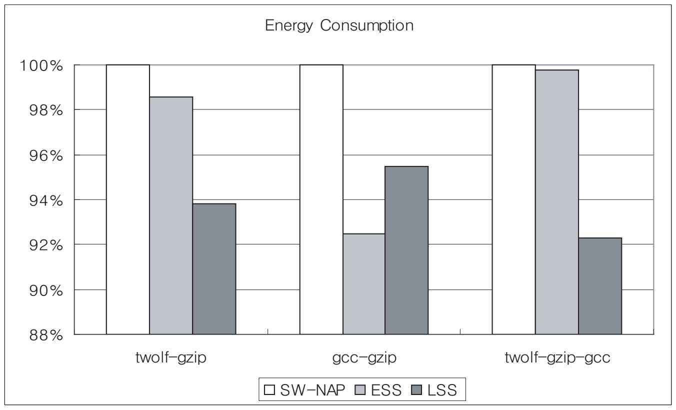


Fig. 6. Energy Consumption in Use of ESS and LSS

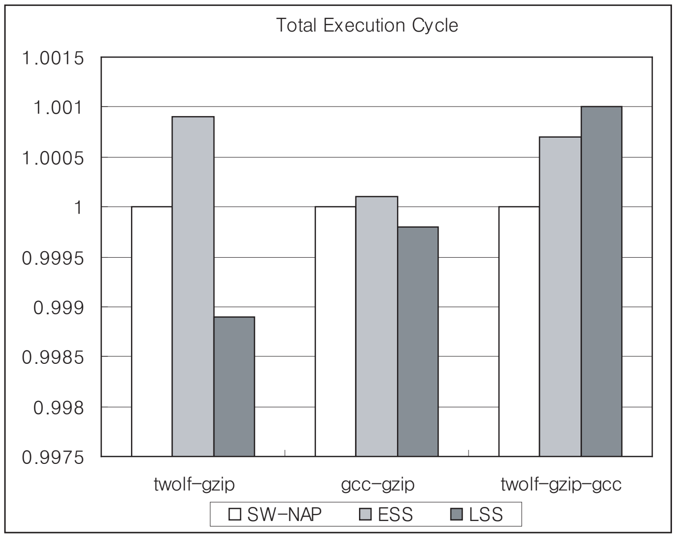


Fig. 7. Total Execution cycles

7 the difference among the scheduling policies were less than 0.1%. By this we can tell that both ESS and LSS increase the energy efficiency of SW-NAP without the prolongation of task execution time.

5. Conclusion

This work pointed out the weakness of the existing approaches on power-aware memory management which tried to raise spatial locality of memory allocation. Existing algorithms are expected not to work well on actual systems due to the unexpected activation of memory blocks from cache eviction.

We suggested a simple and novel power-aware memory management scheme exploiting temporal locality. SW-NAP deactivates all the memory blocks after every tick. We also suggested two scheduling policies to increase the effectiveness of SW-NAP. ESS expedites context switching when the currently running task produces cache misses in the some predefined time interval which are usually the last half of its given ticks. On the contrary LSS extends the execution when the current running task generates no cache miss.

The suggested schemes were implemented on Linux kernel. Evaluation were done for three benchmark programs from SPECCPU 2000 benchmark suite. The energy consumption and the execution time was measured with the modified Bochs emulator. The evaluation results showed that SW-NAP performs better than PAVM by saving about half energy of PAVM. Comparing to PMU which is a hardware approach, SW-NAP consumes about 20% more energy in general. Moreover with ESS and LSS which are the scheduling policies considering memory access patterns, SW-NAP was improved by up to 7% of energy saving compared to SW-NAP without them. Considering that PMU requires dedicated hardware which means increased cost, we conclude that SW-NAP is a satisfiable candidate for the energy efficient memory management.

This work showed the potential of the software based approach exploiting temporal locality for power-aware memory management. The short transition times among the different power modes which were provided by modern memory technologies aid this approach. However, the suggested scheme may affect the fair sharing of processor resource among tasks. And the algorithm is rather intuitive and not intelligent that means a margin to be optimized. If an operating system knows the memory reference patterns precisely by monitoring profiler registers which are provided by many modern processors it can choose the optimal policy for the situation among ESS, LSS or some other customized algorithms to get greater energy efficiency with less performance loss.

Acknowledgments

This work was supported by the Korea Science and Engineering Foundation(KOSEF) grant funded by the Korea government(MOST) (No. R01-2006-000-10724-0)

References

1. M. A. Vireda and D. A. Wallach, Power evaluation of a handheld computer, *IEEE Micro*, vol. 23, no. 1, pp. 66–74, 2003.
2. P. Pillai and K. G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, in *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2001, pp. 89–102, ACM.
3. K. Flautner and T. Mudge, Vertigo: automatic performance-setting for linux, *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 105–116, 2002.

4. E. Seo, Y. Koo, and J. Lee, Dynamic repartitioning of real-time schedule on a multicore processor for energy efficiency, *Lecture Notes in Computer Science*, vol. 4096, pp. 69–78, 2006.
5. H. Huang, P. Pilai, and K. G. Shin, Design and implementation of power-aware virtual memory, in *Proceedings of Usenix Annual Technical Conference*, 2003.
6. Bochs, Bochs, http://bochs.sourceforge.net/.
7. A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, Power aware page allocation, *ACM SIGOPS Operating Systems Review*, vol. 34, no. 5, pp. 105–116, 2000.
8. V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, Scheduler-based dram energy management, in *Proceedings of Design and Automation Conference*, 2003.
9. J. L. Hennessy and D. A. Patterson, *Computer Architecture : A Quantitative Approach, 3rd Ed.*, Morgan Kaufmann Press, 2003.
10. M. Lee, E. Seo, J. Lee, and J. Kim, PABC: Power-aware buffer cache management for low power consumption, *IEEE Transactions on Computers*, vol. 56, no. 4, pp. 488–501, 2007.
11. H. Huang, K. G. Shin, K. Rajamani, T. Keller, E. V. Hensbergen, and F. Rawson, Cooperative software-hardware power management for main memory, in *Proceedings of 4th Workshop on Power-Aware Computing Systems*, 2004.
12. Standard Performance Evaluation Corporation, SPEC CPU2000, http://www.spec.org/cpu2000/.
13. J. Henning, SPEC CPU2000 memory footprint, http://www.spec.org/cpu2000/analysis/memory/, 2000.