A Scheduling of Periodically Active Rank of DRAM to Optimize Power Efficiency

Xi Li[1, 2], Gangyong Jia[1, 2], Chao Wang[1, 2], Xuehai Zhou[1, 2], Zongwei Zhu[1, 2]
[1] Department of Computer Science and Technology, University of Science and Technology of China (USTC)
Hefei, 230027, China
[2] Suzhou Institute for Advanced Study, USTC, Suzhou, China
{gangyong, saintwc, zzw1988}@mail.ustc.edu.cn; {llxx, xhzhou}@ustc.edu.cn

Abstract

Main memory is expected to grow significantly in both speed and capacity for it is a major shared resource among cores in a multi-core system, which will lead to increasing power consumption. Therefore, it is critical to address the power issue without seriously decreasing performance in the memory subsystem. In this paper, we propose a periodically active rank scheduling (PARS) to optimize power efficiency for multi-core DRAM in smart phones. Our scheduling features a three-step design. First, we partition all threads in the system into groups. Second, modify page allocation policy to achieve threads in the same group occupies the same rank but different bank of DRAM. Finally, sequentially schedule threads in one group after another while only active running group's ranks to retain other ranks low power status. As a result, our scheduling periodically activates one rank after another to optimize memory power efficiency. We implement PARS in Linux 2.6.35 kernel running randomly generated workloads containing single-threaded and multi-threaded benchmark. Experimental results show that PARS can improve both the memory power efficiency 26.8% and performance 4.2% average reducing negligible fairness.

Keywords

Power efficiency; main memory; scheduling; page allocation; group

1. INTRODUCTION

In recent years, chip multiprocessors (CMPs) [1] have become increasingly important and common in the computer industry [2] [3]. Similarly, mostly smart phones have adopted multi-core system, dual-core is mainstream and quad-core is high-end, even eight-core will be the next generation. To cope with the substantial memory demand from these processors, density and speed of memory DRAM subsystem have been increased dramatically. However, such increasing in memory size and speed has led to a significant increase in total energy consumed by the memory. Recent studies show a growing concern for energy problems in memory subsystem and that the memory energy may even surpass energy consumption of the CPU, which is traditionally known to be a primary source for energy consumption in computer systems [4] [5].

Many memory management mechanisms have been considered for servers, desktops, and portable computers [6] [7] [8] [9] [10] [11]. Different memory technologies [12] [13] [14] and hierarchies [15] have been investigated as well. The recent exploration of alternative memory technologies is driven by the development of phase change memory (PCM). PCM offers random access capability and performance comparable to DRAM, and non-volatility of flash memories. Subsequently, those characteristics allow PCM to be efficiently included in memory hierarchy or even replace DRAM to a certain extern. These are mostly involved hardware, but we mainly focus on operating system.

Also, some schedulers according memory occupied and simultaneously combining with memory migration have proposed to optimize memory power efficiency [16] [17]. These methods partition threads into groups according to their occupying memory rank which is the basic unit to control memory power status, one group threads occupying almost the same memory ranks. Based on group, priority schedule threads belonging to the same group sequentially, activating corresponding ranks and shutting down other ranks. Although these ways have advantages of power efficiency, they reduce performance seriously because of frequently migrating data from low power ranks. According to our experiment results, execution time of all threads increase more than 30% average.

In order to tackle imbalance of power efficiency and performance, in this paper, we propose a periodically active rank scheduling. We coordinate page allocation policy with operating system scheduler, which our page allocation policy make threads' occupied memory rank more aggregation and regularly.

PARS feature a three-step design. First, according to the relationship among threads, we partition all threads in the system into groups. For example, threads belonging to the same process are sharing a group. The number of thread groups is equal to the number of the memory ranks. Thread group and memory rank are one-to-one correspondence. Second, according to the group number the thread belonging, allocate page in corresponding memory rank but in different banks. All occupied memory of threads in a group aggregates corresponding memory rank almost without escaping. Finally, priority schedule threads in the same group, after all threads in the same group have run sometime, choosing another. When a group is running, only its memory rank is active, after choosing another, switching active memory rank. So, memory rank

periodically active which can optimize power efficiency without reducing much performance from migrating.

We evaluate PARS both in single-core and multi-core situations with a large set of single-thread and multi-thread workloads. Our base results demonstrate that we can improve 26.8% power efficiency and 4.2% performance average. We also analyze other parameters, such as fairness, degree of aggregation, response time and so on. And we are pleased all these parameters are acceptable.

Specially, this paper makes the following major contributions:

- We firstly coordinate page allocation policy with operating system scheduler to optimize memory power efficiency. Through our page allocation policy, aggregating threads' occupying memory ranks according groups without migrating data when running.
- We improve both power efficiency and performance for multi-core, which disturbs conventional thinking of improved power efficiency mostly based on performance reduced.
- We propose degree of aggregation parameter to indicate the effect of page allocation policy to retain other memory ranks stay low power as long as possible.
- We detailed analyze power efficiency, performance, fairness, thermal, average response time and degree of aggregation parameters of PARS.

2. BACKGROUND AND RELATED WORK

2.1 DRAM System

We briefly describe DRAM memory systems and OS memory management mechanism.

DRAM Organization: modern memory system is usually packaged as DIMMs, each of which usually contains 1 or 2 ranks and 8 banks. A memory system can contain multiple channels, and each channel is associated with 1 or 2 DIMMs. A rank is the smallest physical unit for power management. Banks can be accessed parallel, hence, memory requests to different banks can be served concurrently [18]. Memory device can be in four states – *active standby*, *precharge standby*, *active power-down and precharge power-down* – listed in a decreasing order of power dissipation [16].

OS Memory Management: Nowadays, Linux kernel's memory management system uses a buddy system to manage physical memory pages. In the buddy system, the continuous 2^{order} pages (called a block) are organized in the free list with the corresponding order, which ranges from 0 to a specific upper limit. When a program accesses an unmapped virtual address, a page fault occurs and OS kernel takes over the following execution wherein the buddy system identifies the right order free list and allocates on block (2^{order} physical pages) for that program. Usually the first block of a free list is selected but the corresponding physical pages are undetermined [19].

2.2 Related work

There are a number of related studies.

Thread Scheduling. Scheduling algorithms PPT proposed in [16] aimed to active partial memory ranks according running threads to optimize memory power efficiency. This method can improve memory power efficiency, but seriously reduce performance because of migrating data from low power memory ranks. Our previous studies [20] [21] have proposed group scheduling to improve power efficiency and performance as well.

Row Buffer Optimization. In [22], frequently accessed data of different rows are dynamically migrated into row buffer, which can improve the row buffer usage and performance; power consumption is also lowered by reducing the operations of percharge and active.

Thread-based Memory Scheduling. Memory controllers are designed to distinguish the memory access behavior at thread-level [23] [24], so that scheduling modules can adjust their scheduling policy at the running time. TCM [23], which dynamically groups threads into two clusters (memory intensive and non-intensive), and assign different scheduling policy to different group, is the best scheduling policy, which aim to address fairness and throughput at the same time.

Bank Level Partition. BPM [25], which partitions banks according threads to assign different group of banks to different threads to eliminate the interference between threads and improve the overall system performance.

PARS. To the best of our knowledge, this is the first work that improve both power efficiency and performance through coordinating operating system scheduling with page allocation of optimizing both rank-level aggregation and bank-level parallel.

3. PERIODICALLY ACTIVE RANK SCHEDULING (PARS)

3.1 Overview of PARS

Our PARS combines page allocation with the operating system scheduling to improve both power efficiency and system performance, which is designed into three following steps.

3.2 Group Partition

In section 2.1, we have introduced memory rank is the smallest physical unit for power management. We can aggregate current running threads' memory into a memory rank, low power other ranks to reduce memory power. The threads of aggregated into the same memory rank are formed a group. Based on the number of ranks, we partition all threads into the same number groups. Generally, a DRAM has 4 or 8 ranks, so we have 4 or 8 groups.

With multi-core processor for mobile phones being more and more popular, more and more emerging applications are believed to be highly parallel [21]. One application creates more and more threads to parallel finish job. All threads belonging to an application are sharing memory, which we can take advantage to reduce frequency replacing TLB and cache when switching threads. We partition all threads of an application into the same group. And all threads of an application are listed sequentially. Specially, all kernel threads are partitioned into a unique group. Mostly, there are more applications than ranks, so applications are partitioned into different groups. In this paper, we partition applications according to load balance. All threads in the same group are

listed according applications. The details of group partition are shown in Algorithm 1. Each time creating a new thread we all apply this algorithm.

Algorithm 1: group partition

After creating a new thread *T*, *T* ϵ *A*, *A* is an application, *G* is the group of all kernel threads
begin
1: whether the application *A* is already existing in system
2: **if** *T* is kernel thread **then**
3: insert *T* into the back of group *G*;
4: return;
5: **else if** *A* is already existing **then**
6: find group G_1, $A \epsilon G_1$;
7: insert *T* into the back of *A*;
8: return;
9: **else** *A* is a new one **then**
10: find the group of lightest load, G_2;
11: insert *A* into the back of group G_2;
12: insert *T* into the back of *A*;
13: *A* is partitioned into group G_2;
14: return;
End

In our system, all threads of the same application are listed sequentially, all applications in the same group are listed sequentially. Figure 1 demonstrates the list of one group.

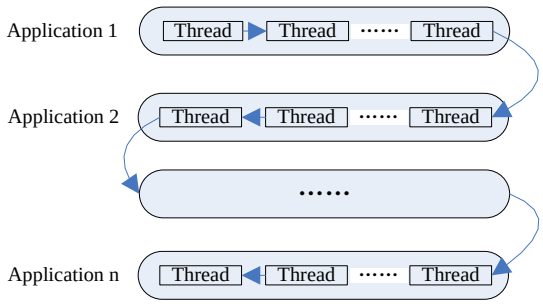


Figure 1 example of one group list

3.3 Page Allocation

Usually, buddy system allocates the first block of a free list to the request thread, so a thread's occupying memory spans all ranks of the memory. And each rank contains multiple physically independent banks which can serve memory requests concurrently and independently, generally, one rank contains 4 or 8 banks. In order to aggregate each thread's memory into single rank and improve request parallelism, we adjust buddy system to manage physical memory pages adding memory rank and bank information.

Figure 2 demonstrates the organization of physical memory pages management of buddy system. The process has briefly analyzed in section 2.1. Figure 3 shows the organization of our physical pages management which adds the rank and bank information. Specially, for the DRAM architecture, there may be no 512 or smaller consequent blocks in our organization. But this is not a big deal for most requests are single page. Each rank/bank organizes its own physical pages.

When a thread accesses an unmapped virtual address, a page fault occurs and OS kernel takes over the following execution which algorithm 2 demonstrates. And each rank has *B* banks.

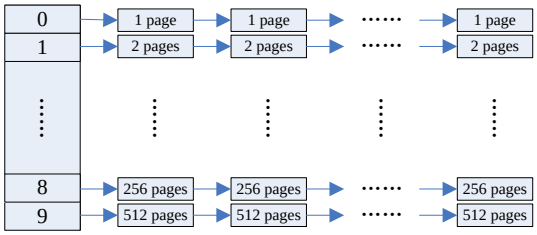


Figure 2 physical pages management of buddy system

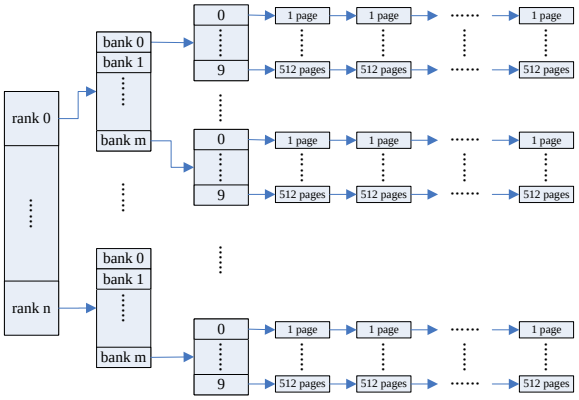


Figure 3 physical pages management of our system

Algorithm 2: page allocation

Thread *T* accesses an unmapped virtual address, OS kernel allocates pages
begin
1: find the group *G* which *T* ϵ *G*;
2: according to the id of *G*, find corresponding rank, *R*;
3: calculate *Bi* = *Tid* % *B*, *Tid* is the thread id of *T*;
4: identify the right order free list of *Bi* in *R*;
5: allocate on block for *T*;
6: return;
End

The difference between our page allocation with buddy system is we add the process of finding group, corresponding memory rank and bank. The cost of the added operation is negligible after optimizing.

With our page allocation policy, one group threads' occupying memory almost aggregates one rank, threads in the same group occupies different banks, which both improve aggregation and parallelism.

3.4 Scheduling and Rank Management Policy

3.4.1 design the PARS

All threads in the system are partitioned into groups, each group occupies only one rank. When a group threads running, only corresponding rank needs to be active, others can be low power. So we coordinate group scheduling [21] with memory rank status management to optimize memory power efficiency.

Figure 4 demonstrates the periodically scheduling threads according group and active memory rank. All threads of an application are listed sequentially, and applications of the same group are listed sequentially. Our PARS has following

periodically phases to choose the next running thread and manage memory rank status:

1) If there are some threads in the same application to current running thread, choose one as the next running thread, keep memory rank status;
2) Else if there are some applications in the same group to the current running thread, choose one application as the next running application and choose one thread among the application as the next running thread, keep memory rank status;
3) Else pick up another group as the running group, select an application among the group, and choose a thread among the application as the next running thread, activate the rank of next running group and set the rank of current running group low power.

In the figure 4, rank with slash represents being active, and active rank is periodically switching according to running group is periodically switching.

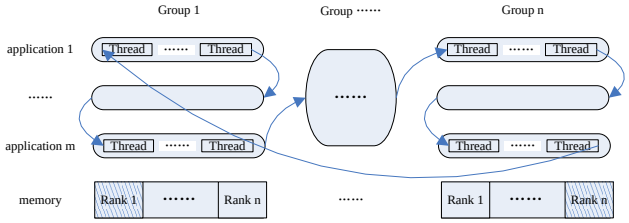


Figure 4 process of threads scheduling and corresponding active rank

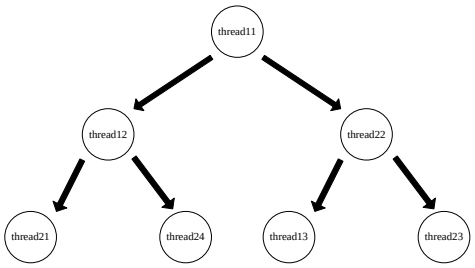
3.4.2 implement the PARS

In the Linux kernel, struct *task_struct* is used to represent each thread. Each thread in the kernel has a unique *task_struct*. Domain *tgid* in the *task_struct* is used to mark which application the thread is belonging to. All threads with the same *tgid* are in the same application according recently thread-process model. Domain *thread_group* in the *task_struct* links the threads with the same *tgid*. We also link different applications according one application threads after another in the same group through domain *thread_group*. Traveling the *thread_group* list can find all threads in the same group. Figure 5(a) shows the default CFS scheduling queue organized into an *rb-tree*. Where each thread is marked with two numbers, the first one indicates the thread group it belongs to, and the second one shows the thread id in the thread group. For example, *thread12* represents a thread in group 1 with thread id is 2. Figure 5(b) demonstrates that through *thread_group* list, all threads in the same thread group are listed by the circular list on one core. All threads belonging thread group 1 are listed by the white circular list and all thread belonging thread group 2 are listed by the slash circular list. Figure 5(c) demonstrates that all threads in the same thread group are listed among two cores. By default, each core will have a unique scheduling queue.

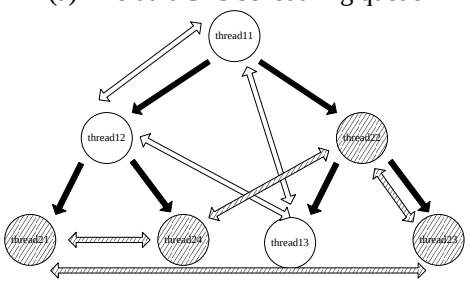
4 EVALUATION

4.1 Simulation Environment

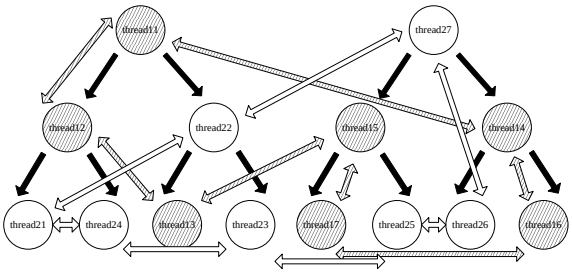
We evaluate the proposed PARS with trace-driven simulation. The virtual memory address trace is obtained by Cachegrind, which is the cache simulation component of Valgrind [26] profiling tool. We use the cycle-accurate DRAM simulator, DRAMsim [27], to model the DRAM system. We annotate the memory traces with correct physical addresses and timestamps according to the page faults and thread scheduling policies. We also model page faults and their associated memory accesses, and reflect the delay caused by memory contention, thread scheduling, and page faults by adjusting the timestamps in traces. To evaluate DRAM temperature, we adopt HotSpot 3.0 [28], which calculates temperature by modeling physical properties. Table 1 shows the processor and memory configurations. From single-threaded application of SPEC2000 benchmark and multi-threaded application of sysbench benchmark, we both select some benchmarks.



(a) Default CFS scheduling queue



(b) All threads in the same thread group are listed together on one core



(c) All threads in the same thread group are listed together among cores

Figure 5 threads are listed according thread group

4.2 Experimental Results

4.2.1 degree of aggregation

In order to reflect the effect of page allocation in aggregating memory, we propose a parameter called degree of aggregation. Degree of aggregation is the average request on the same memory. We define the following formula:

Degree of aggregation = total request / switch times

Total request represents the total numbers of request memory, switch times represents times of switching between

accessing two different ranks. For example, we sequentially list a rank numbers of accessing memory, 1, 1, 2, 3, 5, 3, 3, 4. Total request is 8, switch times is 5. The 5 times contains the second 1 to 2; 2 to 3; 3 to 5; 5to 3; third 3 to 4. Degree of aggregation is 8/5. The bigger degree of aggregation, the better effect of page allocation in aggregating memory.

Table 1 processor and memory configurations

Parameters	Value
Processor	4-core, 2GHz
Memory	2 channels, 2 DIMMs/channel, 2 ranks/DIMM
Memory rank	8 banks/rank

Table 2 shows the comparing default method, PPT with our PARS in degree of aggregation. PPT is the method proposed in [16]. From the table, we can see PARS is much better than other two methods. From the memory request list, we can obviously find PARS prolong more time accessing on one rank, and reduce much more switch times than other two methods.

Table 2 compare the degree of aggregation

	Default method	PPT	PARS
Degree of aggregation	10.6	27.1	35.4

Because our PASR only adjust non-real time threads' scheduling, preserving real-time threads can seize other threads. The degree of aggregation will much bigger if ignoring the disturb of the real-time threads for our PASR. How to remove the disturb of the real-time threads is our future work.

4.2.2 power reducing of the PARS

PARS periodically activates one of the ranks according running group, but each time only one memory rank is active except apply a big continuous block which outspace one rank can supply. Also, our PARS prolongs much more time accessing one the same rank which reduces frequency of switching between ranks. Some kernel threads are real-time threads and always disturb the running according group, which leading to the frequence switch between ranks of kernel threads and others. This switching reduce performance, and even increase power. In order to improve performance, we set the rank of kernel threads active all the time, and periodically activate other ranks according running group.

Figure 6 demonstrates the power consumption of PPT and our PARS comparing with default method. Our PARS can reduce more than 26% power comparing to default method, and also much better than PPT.

4.2.3 performance of PARS

In order to improve parallel, reduce the interference among threads because one bank may receive memory requests from different cores, which probably have different memory access characteristics, and lower the cost of switching between threads, our PARS optimizes in the following parts:

1) partition threads of an application into the same group and priority schedule threads belonging to the same application. The cost of switching between the same application threads is much smaller for sharing the memory address space;
2) allocate pages of different banks for threads in the same group. Memory request from different cores almost access different banks, so seldom interfer among threads from different cores and improve parallel.

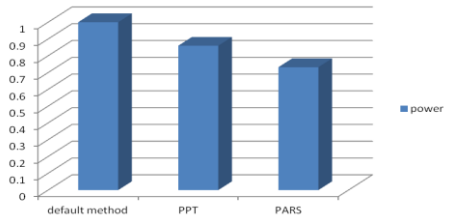


Figure 6 power consumption comparing

Besides improving power efficiency, our PARS improves performance. Figure 7 illustrates performance comparing among three methods. Although improving power efficiency, PPT decreases performance comparing to the default. Our PARS optimizes both in power efficiency and performance.

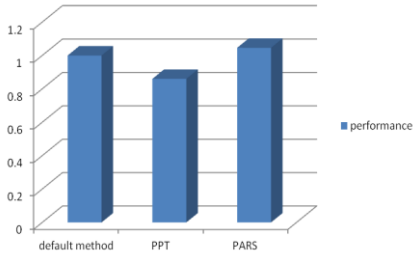


Figure 7 performance comparing

Table 3 demonstrates average overhead of cache and TLB comparing between PPT and PARS. Obviously, our group partition according applications is effective, which reduces cache misses and TLB misses.

Table 3 overhead comparing

	PPT	PARS
L2 cache miss rate	0.094%	0.013%
DTLB misses	26992646	26948934
ITLB misses	17895	12312
ITBL flushes	66	43

Table 4 illustrates average row buffer miss rate comparing among different methods. Apparently, our page allocation according bank is also very effective, which intensely reduces row buffer miss rate.

Table 4 row buffer miss rate comparing

	Default method	PPT	PARS
Row buffer miss rate	58.2%	60.7%	31.3%

4.2.4 fairness of PARS

We use the definition of fairness proposed in [20]. Figure 8 demonstrates the normalized average fairness among threads. The smaller of the value is, the better fairness is. From the figure, we can easily find our PARS are more fairness than PPT, and similar to the default.

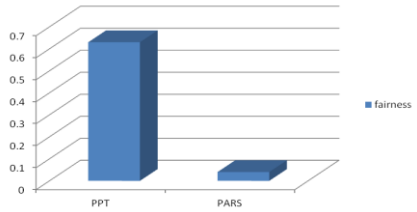


Figure 8 fairness comparing

4.2.5 thermal of PARS

Table 5 illustrates the normalized average peak temperature. In the table, we can find PARS and PPT are better in peak temperature. Both PARS and PPT periodically activate one rank and turn down other ranks. After a rank turning down, its temperature will decrease. Periodically decrease temperature, not always keeping active can reduce peak temperature. Our PARS is better than PPT because of eliminating migration data from low power ranks.

Table 5 Normalized average peak temperature

	Default method	PPT	MAS
Peak temperature	85.9	76.1	75.8

5 CONCLUSION

In this paper, we propose a periodically active rank scheduling (PARS) to optimize power efficiency for multi-core DRAM in smart phones. Our scheduling features a three-step design: group partition, page allocation, and periodically active rank scheduling. According to application, we partition threads from a same application into the same group and list them sequentially. According to rank and bank, we manage physical memory page and allocate page based on group and threads' id, urgently aggregating memory of the same group and eliminating the interference among threads from different cores. According to group, we priority schedule threads in the same group, which prolongs access one same rank and keeping other ranks low power. Experimental results show our PARS can both optimize power efficiency and performance with negligible fairness losing, other parameters such as degree of aggregation and thermal are also well.

6 ACKNOWLEDGMENT

This work is supported by the National Science Foundation of China under grants (No. 61272131, No. 61202053), Jiangsu provincial Natural Science Foundation (No. SBK201240198), Jiangsu production-teaching-research joint innovation project (No.BY2009128). We also gratefully acknowledge the support of our industrial sponsor, SAMSUNG (CHINA) R&D CENTER.

7 REFERENCES

[1] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *ACM SIGOPS Operating Systems Review*, 30, 1996.

[2] S. Y. Borkar. Platform 2015: Intel processor and platform evolution for the next decode. *Intel White Paper*, 2005.

[3] Intel Corporation. Intel's tera-scale research prepares for tens, hundreds of cores, 2006.

[4] L. Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12): 33-37, 2007.

[5] G. Dhiman, R. Ayoub, and T. Rosing. Pdram: a hybrid pram and dram main memory system. *DAC*, pages 464-669, 2009.

[6] H. Huang, P. Pillai, and K. G. Shin. Design and implementation of power-aware virtual memory. *In ATC*, 2003.

[7] M. Lee, E. Seo, J. Lee, and J.-s. Kim. Pabc: Power-aware buffer cache management for low power consumption. *IEEE Transactions on Computer*, 2007.

[8] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini. Dma-aware memory energy management. *In HPCA*, 2006.

[9] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller. Improving energy efficiency by making dram less randomly accessed. *In ISLPED*, 2005.

[10] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *Computer*, 2003.

[11] X. Li, Z. Li, Y. Zhou, and S. Adve. Performance directed energy management for main memory and disks. *Transactions on Storage*, 2005.

[12] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. *In ISCA*, 2009.

[13] W. Zhang and T. Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architecture. *In PACT*, 2009.

[14] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. *In ISCA*, 2009.

[15] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *In ISCA*, 2009.

[16] C. –H. Lin, C. –L. Yang, and K. –J. King. PPT: Joint performance/power/thermal management of dram memory for multi-core systems. *ISLPED*, pages 93-98, 2009.

[17] R. Ayoub, K. R. Indukuri, T. S. Rosing. Energy Efficient Proactive Thermal Management in Memory Subsystem. *ISLPED*, 2010.

[18] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM system. *In ISCA-35*, 2008.

[19] S. Cho, and L. Jin. Managing Distributed, shared L2 Caches through OS-Level page Allocation. *In MICRO-39*, 2006.

[20] G. Jia, X. Li, C. Wang, X. Zhou, Z. Zhu. Memory Affinity: Balancing Performance, Power, Thermal and Fairness for Multi-core Systems. *IEEE Conference on Cluster Computing*, 2012.

[21] X. Li, G. Jia, Y. Chen, Z. Zhu, X. Zhou. Share Memory Aware Scheduler: Balancing Performance and Fairness. *ACM/IEEE the 22th Great Lake Symposium on VLSI (GLSVLSI)*, 2012.

[22] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-Pages: Increasing DRAM Efficiency with Locality-Aware. *In ASPLOS*, 2010.

[23] Y. Kim, M. Papamicheal and O. Mutlu. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. *In MICRO-43*, 2010.

[24] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. *In HPCA-16*, 2010.

[25] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, C. Wu. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. *In PACT*, 2012.

[26] C. Armour-Brown, K. Fitzhardinge, T. Hughes, N. Nethercote, P. Mackerras, D. Mueller, J. Seward, R. Walsh, and J. Weidendorfer. Valgrind. *http://valgrind.org/*, 2000-2007.

[27] D. Wang, B. Ganesh, and B. Jacob. The university of Maryland memory-system simulator. *http://www.ece.umd.edu/dramsim/*, 2006.

[28] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. *Proceedings of the 30th International Symposium on Computer Architecture*, pages 2-13, 2003.