Coordinate Page Allocation and Thread Group for Improving Main Memory Power Efficiency

Gangyong Jia[1, 2], Xi Li[2], Jian Wan[1], Liang Shi[3], Chao Wang[2]
[1] Department of Computer Science and Technology, Hangzhou Dianzi University
Hangzhou, 310018, China
[2] Department of Computer Science and Technology, University of Science and Technology of China (USTC)
Hefei, 230027, China
[3] Department of Computer Science and Technology, Chongqing University
Chongqing, 310018, China
{gangyong, saint}@mail.ustc.edu.cn; jianwan@hdu.edu.cn; llxx@ustc.edu.cn

Abstract
Main Memory is responsible for a large and increasing fraction of the energy consumed by multi-core systems. Therefore, it is critical to address the power issue in the memory subsystem. In this paper, we present a solution to improve memory power efficiency through coordinating page allocation and thread group scheduling (CAS). Partitioning all threads into different thread groups, after using proposed page allocation, threads in the same thread group occupy the same memory rank. Adjusting default Linux CFS, implement thread group scheduling. The CAS alternates active partial memory periodically to allow others power down and prolongs the idleness parts. Our experimental results show that this approach improves energy saving by 10% and reduces performance overhead by 8% with respect to the state of the art polices.

Categories and Subject Descriptors:
D.4.1 scheduling
General Terms:
Management, Performance, Design.
Keywords:
Main memory; page allocation; thread group scheduling; power efficiency; performance

1. INTRODUCTION
Historically, within the server, the processor has dominated energy consumption. However, as processors have become more energy-efficient and more effective at managing their own power consumption, their contribution has been decreasing. In contrast, main memory energy consumption has been growing [1, 2, 3], as multi-core systems are requiring increasing main memory bandwidth and capacity. Making matters worse, memory energy management is challenging in the context of servers with modern (DDR*) DRAM technologies. Today, main memory accounts for up to 40% of server energy [1]—comparable to or slightly higher than the processors' contribution. In reality, the fraction attributable to memory accesses may be even higher [4].

Recent works have considered reducing the number of DRAM chips that are accessed at a time (rank subsetting) [5, 6] and even changing the microarchitecture of the DRAM chips themselves to improve energy efficiency [7]. A common theme of these works is to reduce the number of chips or bits actually touched as a result of a memory access, thereby reducing the dynamic memory energy consumption. However, Rank subsetting requires changes to the architecture of the memory DIMMs (Dual In-Line Memory Modules), which are expensive and increase latency. Changing DRAM chip microarchitecture may have negative implications on capacity and yield [4].

Also, some research has proposed power aware page allocation [8] and scheduling [10, 17, 18, 19, 20] separately which focus on creating memory idleness to reduce memory power consumption. But they are difficult in creating enough idleness without excessively degrading performance.

Thus, this paper coordinates page allocation and thread group scheduling (CAS) to improve main memory power efficiency. According sharing memory address space and load balance, all threads in the systems are partitioned into thread groups. Modifying operating system page allocation, allocate memory pages in the same memory rank (memory rank is the smallest unit for power management) to threads in the same thread group. Threads of the same group are scheduled simultaneously. Running threads in the same group only occupy one rank, all other ranks can power down.

Specifically, this paper makes the following major contributions:

1) Through coordinating page allocation and thread group scheduling, prolong memory idleness, which creates more idleness without excessively degrading performance;

2) Based on sharing memory address space to partition threads, decrease switch overhead between two threads in the same group;

3) According running thread group to manage memory rank power modes, reduce the frequency of all rank power modes transition;

We compare the propose CAS with some state of the art policies, the experimental results we report show that our approach improves power efficiency by 10% and reduces performance overhead by 8%.

The rest of this paper is organized as follows. Section 2 introduces the basics of DRAM organization. Section 3 explains our coordinating page allocation and thread group scheduling platform. Section 4 describes experimental methodology and section 5 presents the results of our

This work is supported by the National Science Foundation of China under grants (No. 61272131, No. 61202053, No. 61003077, No. 61202094). Jiangsu provincial Natural Science Foundation (No. SBK201240198), Jiangsu production-teaching-research joint innovation project (No.BY2009128).

HotPower'13, November 03 - 06 2013, Farmington, PA, USA

experiments. We discuss the related work in section 6. Finally, section 7concludes this paper.

2. BACKGROUND

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 [11]. Figure 1 demonstrates one organization of a modern memory subsystem. 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 [10].

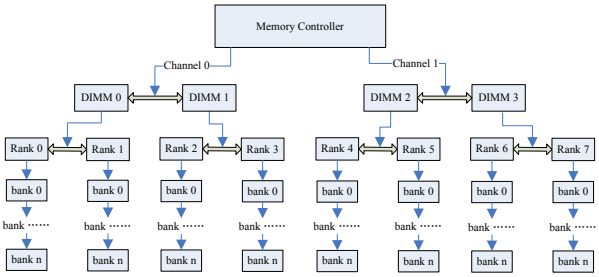

Figure 1 organization of a modern memory subsystem

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 [12].

3. CAS FRAMEWORK

The main idea of the proposed CAS framework is to orchestrate page allocation and thread group scheduling to improve power efficiency. Figure 1 shows our CAS configuration for a 2-channel memory system where each channel contains four ranks. In this paper, threads and ranks are partitioned into 4 groups, all kernel threads partitioned into one group, and all others partitioned into 3. Each group uses two ranks associated with two different channels. Threads of the same group are scheduled simultaneously. Threads in different groups are scheduled in round-robin fashion. Only two rank groups are active at each scheduling interval, one for kernel group and the other for running user group. The memory ranks of non-active groups could be turned into the low-power mode to save energy.

Different DRAM organization affects the power and performance of the memory system. When the memory system is partitioned into more groups, ranks and channels allocated to each group are fewer; that means more memory ranks could be shut down to achieve more power savings, which in turn reduces available bandwidth.

3.1 Thread Group Partition

Because of the 4 memory rank groups, we partition all threads into 4 groups based on sharing memory address space and load balance.

Firstly, all threads sharing memory address space are partitioned into the same group. Switching between threads sharing memory address space in the scheduling avoids replacing the TLB and cache, which has advantage in performance improvement. Specially, we partition all kernel threads into a unique group. And the two ranks allocating to kernel group is active all the time.

Secondly, because only based on sharing memory address space partitioned groups are much more than 3, we partition them into 3 groups according load balance.

So, all threads are partitioned into 4 groups, which one group contains all kernel threads, all non-kernel threads are divided into other 3 groups.

Figure 2 demonstrates the process of a new thread partitioned into one thread group. After a new thread is emerging, confirm there are some threads sharing the same memory address space with this new thread. If existing, find the group those threads belonging to. Else, find a group according load balance. Add the new thread into the group.

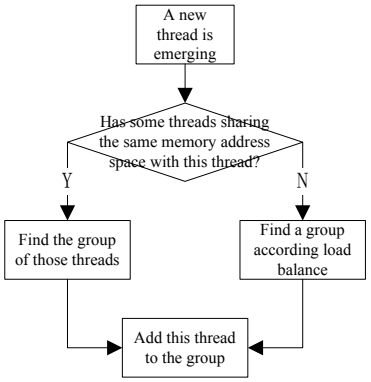

Figure 2 the process of a new thread partitioned into one group

3.2 Page Allocation

In Linux operating system, the default page allocation is using buddy algorithm, which allocates the first block of a free list to the request thread. So, a thread's occupying memory may cover all ranks of the memory. Buddy algorithm takes advantage of parallelism to improve performance. However, the necessary amount of banks one program requires is limited [13].

In contrast, our page allocation focuses specifically on maximizing energy efficiency. Considering the smallest power management unit rank, our algorithm allocates physical memory page to a thread based on belonging group, threads of the same group occupy the same rank group. So, a thread's pages aggregate in two ranks, spreading all banks of these two ranks, which improves power efficiency and prevents performance degradation simultaneously.

Figure 3 demonstrates our organization of physical memory page. The difference between ours with the default is our organization adds the rank information. Each rank group has free block lists, which likes whole memory partitioning into 4 in this paper. Every time requiring a free block, firstly determine the rank group, and then allocate corresponding free block.

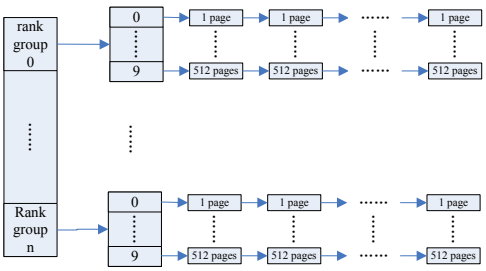

Figure 3 our organization of physical memory page

Algorithm 1 describes how our page allocation algorithm allocates a free page after requiring. The difference of our page allocation algorithm to the default buddy's is we restrict the pages of every thread required into its rank group, not spreading among whole memory. Therefore, our algorithm prolongs more idleness and retains enough parallelism.

Algorithm 1: our page allocation algorithm

Thread *T* accesses an unmapped virtual address, OS kernel allocates pages
begin
1: find the group *G* which $T \epsilon G$;
2: according to the id of *G*, find corresponding rank group, *R*;
3: find the free lists of *R*;
4: search the suitable free block based on buddy algorithm within rank group *R*;
5: allocate a block for *T*;
6: return;
End

3.3 Thread Group Scheduling

After all threads are partitioned into groups, we rearrange these threads according group, showing in figure 4. Threads in one group are organized into an *rb-tree* according each thread's *vruntime*, which is the same with the default Completely Fair Schedule (CFS) of Linux. 4 groups are also organized into an *rb-tree*, and their respective location is based on group *vruntime*. We define the group *vruntime* is the sum of all its threads' *vruntime*.

$$G_{vruntime} = \sum_{i=1}^{n} vruntime_i \qquad (1)$$

$G_{vruntime}$ and $vruntime_i$ represents the *vruntime* of group *m* and *vruntime* of thread *i* respectively. Our rearrangement forms a two-level *rb-tree.*

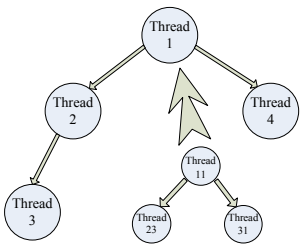

Figure 4 arrange all threads according group

After getting each group's *vruntime*, group is becoming the unit of getting CPU time in our scheduling. When a group running more time than obtained, switch to the next group. Also, threads obtain CPU time within a group according *vruntime*. Therefore, our CAS forms a two-level scheduler. The first level is group schedule, which is the smallest unit seen in the operating system. The second level is thread schedule within group, which also uses CFS policy.

Figure 5 demonstrates the process of our two-level CAS. As we know, all scheduler focuses on how to choose the next running thread. In our CAS, if current running group occupies less time than its allocation, choose the *leftmost* thread in the current *rb-tree* organized by threads. If current running group occupies more time than its allocation, choose the *leftmost* group in *rb-tree* organized by groups and choose the *leftmost* thread in the current *rb-tree* organized by threads of the *leftmost* group.

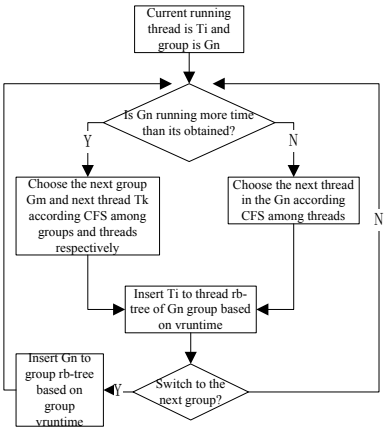

Figure 5 the process of our CAS

3.4 Memory Mode Control Policy

Memory mode control in the CAS framework is archieved through group switching. Match to the periodically scheduling threads in different groups, memory rank groups also periodically active and power down. Because it takes several microseconds to perform power down, which consumes power and time, therefore, in order to maximize energy efficiency, it must prevents switching frequently. So, the question remains to be answered is how long the group switch interval should be. As we know, the shortest scheduling period (*sum_runtime*) is 20ms in Linux, and so the shortest switch interval is 20/3 ms (because we only partition 3 non-kernel groups). 20/3ms is much longer than microseconds, so the cost of rank mode switching is negligible. Therefore, setting rank mode switch according to thread group switch could achieve power savings while preventing high overhead from frequently switch. The more threads in the system, the more idleness can prolong.

4. EXPERIMENTAL SETUP

We use MARSSX86 [14] as the base full-system architectural simulator to run Linux 2.6.31 and extend its memory part using DRAMSim simulator to simulate DDRx DRAM systems in the details. Table 1 shows the major simulation parameters. To estimate the power consumption of DRAM devices, the DRAMSim simulator keeps tracking the states of each memory channel, rank and bank. It follows the Micron power calculation methodology by default [15]. The parameters used to calculate the DRAM power and energy are the same with [4].

Table 1 Processor and memory configurations

Feature	value
CPU cores	Quad core, 2.4GHz
L1 I/D cache (per core)	16KB, 2-way
L2 cache (shared)	64KB
Cache block size	64bytes
Memory configuration	2GB, 2channels, 8ranks, 8banks per rank

In order to evaluate our CAS, we simultaneously run different combinations of selected from sysbench [16] and SPEC2006. In table 2, the *number-appname* notation is the number of threads of the application with the name of

appname for sysbench; for SPEC2006 workload, it is the number of copies of the application with the name of *appname*.

Table 2 workload descriptions

Mixes	sysbench, SPEC2006
mix1	12-sysbench cpu, 8-omnetpp
mix2	12-sysbench memory, 8-omnetpp
mix3	6-sysbench cpu, 6-sysbench memory, 8-omnetpp
mix4	24-sysbench cpu, 16-omnetpp
mix5	24-sysbench memory, 16-omnetpp
mix6	12-sysbench cpu, 12-sysbench memory, 16-omnetpp
mix7	48-sysbench cpu, 32-omnetpp
mix8	48-sysbench memory, 32-omnetpp
mix9	24-sysbench cpu, 24-sysbench memory, 32-omnetpp

5. EXPERIMENTAL RESULTS

In this section, we first examine if the CAS policy prolongs more idleness to power aware page allocation policy. We then show how the proposed CAS controls memory power through group switching. Finally, we detailed analyze the different aspects of performance influence in our CAS.

5.1 Idleness Evaluation of CAS

One of the most important thing to reduce memory power consumption is to create enough memory idleness. But they are difficult because of either causing excessively overhead or degrading performance for recent works. In order to evaluate our CAS in prolonging how much idleness, we propose a metric *ior* (idleness time of each rank), which represents the average idle time of each rank after entering into low power mode.

$$ior = \frac{\sum_{i=1}^{n} ior_i}{n} \qquad (2)$$

ior_i represents the average idle time of rank i after entering into low power mode.

$$ior_i = \frac{\sum_{k=1}^{m} Ti_k}{m} \qquad (3)$$

Ti_k represents each idleness time of rank i. The bigger of the *ior* metric, the more idleness are created.

Figure 6 demonstrates the normalized *ior* value to non-optimization. In this figure, default, PAPA and PPT [10] represents the non-optimization policy in memory power efficiency, power aware page allocation policy, and state of art policy through scheduling respectively. From this figure, we can find PAPA and PPT are creating more idleness than default, and our CAS is better than them. Also, along with the more threads (mix1, 2, 3 are 20 threads, mix4, 5, 6 are 40, and mix7, 8, 9 are 80), the *ior* value are bigger, and means the more idleness are created. More threads in the system mean each group has more threads, which also means each group has more time in each period. Therefore, less switching will happen.

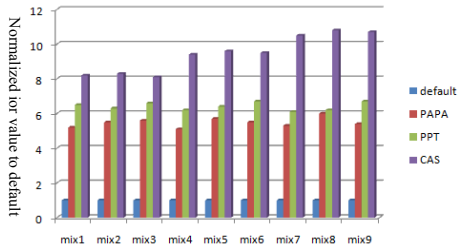

Figure 6 Normalized ior value to default

5.2 Power Efficiency Improving of CAS

Through prolonging more idleness time, figure 7 demonstrates our CAS reduces more memory power under using our group based mode control policy. Similarly, along with the more threads, more power is reduced for more idleness time prolonged. In 20 threads circumstances (mix1, 2, 3), our CAS saves more 9% power than PAPA and 4% than PPT on average; in 40 threads circumstances, more 12% than PAPA and 7% than PPT; in 80 threads circumstances, more 16% than PAPA and 11% than PPT. For whole system, CAS also saves 7.6%, 5.7% and 4.2% power than CFS, PAPA and PPT respectively.

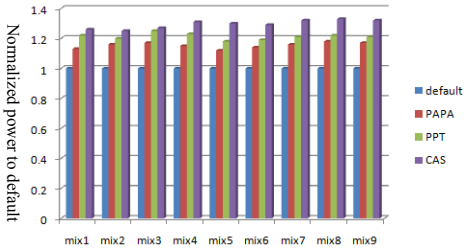

Figure 7 Normalized power to default

5.3 Performance Analysis of CAS

Figure 8 demonstrates the finish time of three policies normalized to default. The longer the finish time, the more performance declined. From this figure, our CAS has better performance than both PAPA and PPT obviously, and the more threads, the better performance our CAS is. In 20 threads circumstances (mix1, 2, 3), our CAS consumes less 5% time than PAPA and 7% than PPT on average; in 40 threads circumstances, less 7% than PAPA and 9% than PPT; in 80 threads circumstances, less 8% than PAPA and 12% than PPT.

Table 3 and table 4 demonstrate the ratio switch between sharing memory address space and each thread occupies average bank number of four policies respectively, which explain the reason for more threads better performance of our CAS. The more ratio of switching between sharing memory address space, the better performance it is. The more each thread occupies average bank number, the more parallelism it is, and the more performance will be. From table 3 and 4, we can easily to know the reason of our better performance than both PAPA and PPT.

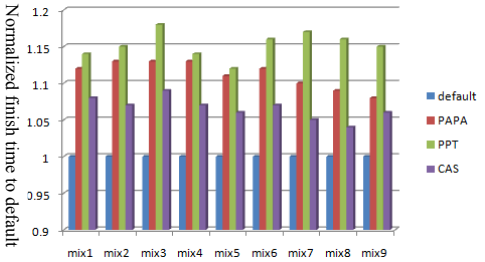

Figure 8 Normalized finish time to default

Table 3 Ratio of switching between sharing memory

	default	PAPA	PPT	CAS
mix1	42%	41%	54%	85%
mix2	43%	45%	58%	84%
mix3	28%	31%	46%	76%
mix4	57%	58%	62%	88%
mix5	54%	56%	61%	90%
mix6	39%	40%	52%	79%
mix7	65%	67%	73%	92%
mix8	67%	62%	75%	93%
mix9	48%	50%	64%	84%

Table 4 each thread occupies average bank number

	default	PAPA	PPT	CAS
mix	18.2	6.8	6.4	14.6

6. CONCLUSION

In this paper, we propose the CAS to improve main memory power efficiency through coordinating page

allocation and thread group scheduling. Our page allocation aggregates required for each group within two ranks, which prevents disturbing too many ranks and retains the parallelism simultaneously. Thread group scheduling reduces the memory mode switching frequency. Our results show that CAS prolongs more idleness to the power aware page allocation policy and scheduling, therefore more power efficiency in memory, but much less performance degradation. Furthermore, CAS is more power efficiency in the more threads system, which shows great promise in future for more and more threads created for a application.

7. REFERENCES

[1] L. A. Barroso and U. H¨olzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Synthesis Lectures on Computer Architecture, Jan. 2009.

[2] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy Management for Commercial Servers. IEEE Computer, 36(12), December 2003.

[3] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. ISCA '09: International Symposium on Computer Architecture, 2009.

[4] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. MemScale: Active Low-Power Modes for Main Memory. In ASPLOS, 2011.

[5] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber. Future scaling of processor-memory interfaces. SC '09 Super Computing, 2009.

[6] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-rank: Adaptive DRAM architecture for improving memory power efficiency. MICRO '08: Symposium on Microarchitecture, Nov. 2008.

[7] A. N. Udipi, N. Muralimanohar, N. Chatterjee, Rajeev Balasubramonian, A. Davis, and N. P. Jouppi. Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores. ISCA'10: International Symposium on Computer Architecture, 2010.

[8] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power Aware Page Allocation. ASPLOS '00: Architectural Support for Programming Languages and Operating Systems, 2000.

[9] X. Fan, C. Ellis, and A. Lebeck. Memory Controller Policies for DRAM Power Management. In Proceedings of the International Symposium on Low-Power Electronics and Design, August 2001.

[10] 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.

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

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

[13] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, Chengyong Wu. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. *In PACT'12*, 2012.

[14] Patel, Avadh, et al. MARSSx86: a full system simulator for x86 CPUs. *In DAC*, 2011.

[15] Micron. –Calculating Memory System Power for DDR3," July 2007.

[16] Kopytov, A. SysBench: a system performance benchmark. http://sysbench.sourceforge.net/index.html. 2004.

[17] Gangyong Jia, Xi Li, Chao Wang, Xuehai Zhou, Zongwei Zhu. Frequency Affinity: Analyzing and Maximizing Power Efficiency in Multi-core System. *IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS),* 2012.

[18] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, Carla Ellis. Power aware page allocation. *In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* 2000.

[19] Gangyong Jia, Xi Li, Chao Wang, Xuehai Zhou, Zongwei Zhu. Memory Affinity: Balancing Performance, Power, Thermal and Fairness for Multi-core Systems. *IEEE Conference on Cluster Computing, Beijing, China,* 2012.

[20] Xi Li, Gangyong Jia, Yun Chen, Zongwei Zhu, Xuehai Zhou. Share Memory Aware Scheduler: Balancing Performance and Fairness. *ACM/IEEE the 22th Great Lakes Symposium on VLSI (GLSVLSI),* 2012.