Received November 30, 2015, accepted December 6, 2015, date of publication December 10, 2015,
date of current version February 29, 2016.

Digital Object Identifier 10.1109/ACCESS.2015.2507576

Dynamic Resource Partitioning for Heterogeneous Multi-Core-Based Cloud Computing in Smart Cities

GANGYONG JIA[1,2], (Member, IEEE), GUANGJIE HAN[3,4], (Member, IEEE), JINFANG JIANG[3,4], NING SUN[3,4], AND KUN WANG[5], (Member, IEEE)

[1]Department of Computer Science, Hangzhou Dianzi University, Hangzhou 310018, China
[2]Key Laboratory of Complex Systems Modeling and Simulation, Ministry of Education, Hangzhou 310018, China
[3]Department of Information and Communication Systems, Hohai University, Changzhou 213022, China
[4]Guangdong Petrochemical Equipment Fault Diagnosis Key Laboratory, Guangdong University of Petrochemical Technology, Guangdong 525000, China
[5]Key Laboratory of Broadband Wireless Communication and Sensor Network Technology, Ministry of Education, Nanjing University of Posts and Telecommunications, Nanjing, 210003, China

Corresponding author: G. Han (hanguangjie@gmail.com)

This work was supported in part by the National Science Foundation of China under Grant 61572172, Grant 61572262, in part by the Natural Science Foundation of Jiangsu Province, China, under Grant BK20140248, and in part by the Zhejiang Provincial of Natural Science Foundation under Grant LQ14F020011.

ABSTRACT As the smart cities emerged for more comfortable urban spaces, services, such as health, transportation, and so on, need to be promoted. In addition, the cloud computing provides flexible allocation, migration of services, and better security isolation; therefore, it is the infrastructure for the smart cities. Single instruction-set architecture (ISA) heterogeneous multi-core processors have higher performance per watt than their symmetric counterparts and are popular in current processors. In current cloud computing, which integrates a few fast out-of-order cores, coupled with a large number of simpler, slow cores, all cores expose the same ISA. The best way to leverage the effectiveness of these systems is to accelerate sequential CPU-bound threads using fast cores, and to improve the throughput of parallel memory-bound threads using slow cores. However, shared hardware resources, such as memory, respond to requests from all cores, which interfere with each other, leading to both low speed for fast cores and low throughput for slow cores. In this paper, we propose a dynamic resource partitioning (DRP) method for single-ISA heterogeneous multi-cores, which partitions the shared resources according to both threads' requirements for the shared resources and the performance of their running cores. The key principle is to profile both threads' resource characteristics at run-time and the performance of the cores that the threads are running on to estimate demands for resources. Then, we use the estimation to direct our resource partitioning. Moreover, we integrate our DRP with current memory scheduling policies to improve the system performance further for the two methods being orthogonal.

INDEX TERMS Single-ISA heterogeneous multi-core, cloud computing, performance per watt, dynamic resource partitioning, memory scheduling, performance.

I. INTRODUCTION

Power density concerns in processor ICs led to the multi-core era [1] where a single and very powerful processor has been replaced by several smaller cores with more modest computational capabilities [2]. As long as the cores are identical, incoming program threads can be assigned to cores arbitrarily by the operating system (OS). However, for a given power budget, symmetric multi-core processors (SMPs) have been shown to be outperformed by single-instruction-set architecture (ISA) heterogeneous multi-core processors (HMPs), which can cater for the needs of diverse workloads [3]–[5], such as the cloud computing.

In cloud computing, multiple virtual machines (VMs) for different services can be collocated on a single physical server, and they can operate independently with virtualization technology, which provides flexible allocation, migration of services, and better security isolation. So the cloud computing is the infrastructure for the smart cities, which provides better

2169-3536 © 2015 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

health service, transportation service and so on. A single-ISA heterogeneous multi-core processor (HMP) includes cores exposing the same instruction-set architecture, but differing in features, size, speed and power consumption [6], [7]. A typical HMP would contain a few complex, large and high-power fast cores and a number of simple, small and low-power slow cores. It is well known that HMP systems can mitigate scalability bottlenecks in parallel threads by accelerating sequential threads on fast cores [6], [8], [9].

However, threads running concurrently contend with each other for the shared resources in multi-core systems, containing both SMP and HMP systems. Threads can be slowed down compared to when it runs alone and entirely owns the shared resources, such as memory system. The problem of shared resources contention is more serious in HMP systems for fast cores being unable to accelerate, because they can't be responded faster from shared resources. So, the effectiveness of the HMP can't be leveraged, and it is wasting of the fast cores. Therefore, if the shared resources contention of the HMP system is not properly managed, it can degrade both individual thread, especially thread running on fast core, and overall system performance, simultaneously causing system unfairness [10]–[16]. In addition, memory streams of different threads are interleaved and interfere with each other at DRAM memory in HMP system; the inter-thread interference destroys original spatial locality and bank level parallelism of individual threads, thus severely degrading system performance [11], [12]. As the number of cores on a chip continues to grow in HMP, the contention for the limited shared memory resources and the interference become more serious [11].

The effectiveness of the HMP system can be evaluated by three objectives: power efficiency, system throughput and fairness. The most important one is the power efficiency, which is the most advantage of the HMP. The second is the overall system throughput, which should remain high. And the last is the fairness, which is no single thread, especially the thread running on fast core, should be disproportionately slowed down.

Previously proposed two kinds' techniques to improve throughput and/or fairness for SMP system: the first one is memory scheduling, which reorders memory accesses. So it can potentially recover a portion of original spatial locality. However, the primary design consideration of the memory scheduling is not reclaiming the lost locality. Furthermore, the effectiveness in recovering locality of memory scheduling is restricted due to the limited scheduling buffer size and the arrival interval (often large) of memory requests from a single thread. In a word, memory access scheduling can somewhat decrease inter-thread interference, but it does not address the issue at its root [17], [29], [30]. The other one is resource partition, mainly memory partition, which divides memory resource among threads to reduce interference. So solves the interference at its root and improves DRAM system performance. However, current memory partitioning policies are proposed for SMP, which is based on the hypothesis of all cores having the same performance. In HMP, the most advantage and challenge is to exert the different characteristics of the heterogeneous performance core. Therefore, current memory partitioning policies don't suit HMP system. Moreover, resource partitioning does not take into account system fairness, and its ability to improve fairness is also restricted.

To maximize the effectiveness of the HMP system, we propose Dynamic Resource Partitioning (DRP) which dynamically partitions shared resources to accommodate the core/thread pair's disparate requirements. The key idea is to profile both cores' and threads' resource characteristics at run-time and estimate their needs for shared resources, mainly shared caches, memory banks and memory bandwidth, then direct our resources partitioning based on the estimation. Moreover, in order to improve fairness, we combine memory scheduling police into our DRP, which can improve throughput and fairness simultaneously to maximize effectiveness of the HMP. Experimental results demonstrate that this combination is able to enhance power efficiency, throughput and fairness simultaneously for HMP system, therefore, maximizing the effectiveness.

In summary, the paper aims to make the following contributions through the proposal of DRP:

1) We propose a dynamic resource partitioning to maximize effectiveness of the single-ISA heterogeneous multi-core system. DRP dynamically partitions shared resources, containing shared cache, memory banks and memory bandwidth, according to core/thread pair's resource requirement.
2) We combine DRP with memory scheduling policies to improve power efficiency, throughput and fairness simultaneously, because resource partitioning is orthogonal to memory scheduling. As resource partitioning focuses on preserving row-buffer locality, memory scheduling focuses on improving system performance by considering core/threads' memory access behavior and system fairness.
3) We estimate the core/thread's resources requirement according to the characteristic of both the core and the thread running on it, not just the thread's characteristic. It is important to character the core's performance in the HMP for maximizing its effectiveness.
4) Experimental results show our DRP is better than current related approaches in maximizing effectiveness of the HMP system.

The rest of this paper is organized as the follows. Section II elaborates on essential background and related work. Section III explains our DRP framework. Section IV describes experimental methodology and Section V presents the results of our experiments. Finally, Section VI concludes this paper.

II. BACKGROUND & RELATED WORKS

In this section, we provide a review of DRAM system and discuss how past research dealt with the challenges of

providing power efficiency, performance and fairness for modern memory systems in HMP.

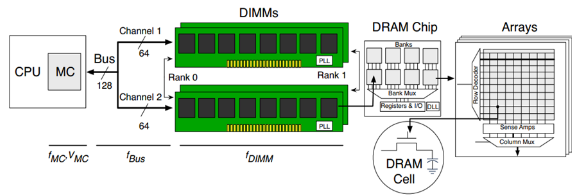


FIGURE 1. Organization of a modern memory subsystem.

A. DRAM SYSTEM

1) DRAM ORGANIZATION

Figure 1 illustrates the multiple levels of organization of the memory subsystem. To service memory accesses, the memory controller (MC) sends commands to the DIMMs on behalf of the CPU's last-level cache across a memory bus. As shown, recent processors have integrated the MC into the same package as the CPU. To enable greater parallelism, the width of the memory bus is split into multiple channels. These channels act independently and can access disjoint regions of the physical address space in parallel [18].

Multiple DIMMs may be connected to the same channel. Each DIMM comprises a printed circuit board with register devices, a Phase Lock Loop device, and multiple DRAM chips. The DRAM chips are the ultimate destination of the MC commands. The subset of DRAM chips that participate in each access is called a rank. The number of chips in a rank depends on how many bits each chip produces/consumes at a time. Each DIMM can have up to 16 chips, organized into 1-4 ranks.

Each DRAM chip contains multiple banks (typically 8 banks nowadays), each of which contains multiple two-dimensional memory arrays. The basic unit of storage in an array is a simple capacitor representing a bit the DRAM cell. Thus, in a $\times 8$ DRAM chip, each bank has 8 arrays, each of which produces/consumes one bit at a time. However, each time an array is accessed, an entire multi-KB row is transferred to a row buffer. This operation is called an activation or a row opening. Then, any column of the row can be read/written over the channel in one burst. Because the activation is destructive, the corresponding row eventually needs to be pre-charged, that is, written back to the array.

2) 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 2order 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 (2order physical pages) for that program. Usually the first block of a free list is selected but the corresponding physical pages are undetermined [19].

B. HMP-POSTED CHALLENGES

As HMP contains a few complex, large and high-power fast cores and a number of simple, small and low-power slow cores, and the core number keeps increasing, HMP architecture poses three major challenges on shared resources:

1) LOW POWER EFFICIENCY

The advantage of the HMP is the higher performance per watt. Therefore, it is the best way to leverage the effectiveness of this architecture is to accelerate sequential CPU-bound threads using fast cores, and to improve throughput of parallel memory-bound threads using slow cores. However, the shared resources restrict its effectiveness. Normally, the fast core needs the fast response from shared resources for preventing waste of CPU time in order to accelerate sequential CPU-bound threads; and the slow core is better to tolerate the long time respond from shared resources. The fast core has more cycles, which can run more instructions, in the same time. Moreover, current shared resources, such as shared cache and memory, don't respond differently to different cores. So the speed of the fast core is restricted by the shared resources, then the power efficiency is limited. As shown in figure 2, there is large disparity between PSO (accelerate sequential CPU-bound threads using fast cores and improve throughput of parallel memory-bound threads using slow cores, but all threads share resources) and the optimal system performance (thread runs alone). The results are from the 4-core platform with 1-fast core and 3-slow core, and the workloads are demonstrated in experimental setup section. Therefore, there is much potentiality in power efficiency for HMP system through shared resources management.

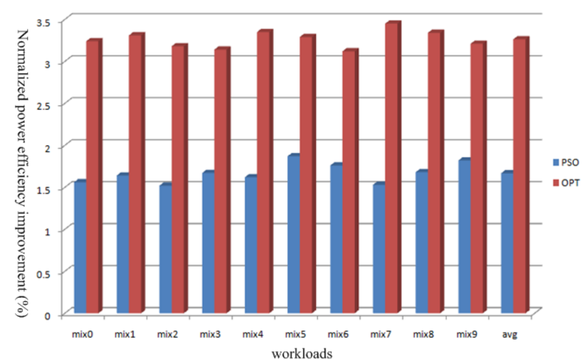


FIGURE 2. Normalized power efficiency improvement.

2) INTERFERENCE

With the core number and thread number keep increasing in the HMP system, one of major challenges is interference, which contains cache interference, memory interference and so on. Use memory interference as an example, usually a single thread's memory requests have good locality and can exhibit good row buffer hit rate. But the locality is

significantly reduced in a multi-core system with running many threads parallel. Therefore, row buffer hit rate decreases sharply, leading to poor overall system performance. Figure 3 demonstrates that the row buffer hit rate decreases significantly with the thread number increased. The y-axis shows the row buffer hit rate, and the x-axis presents the increased parallel running threads. The interference mainly derives from two aspects: the first one is that operating system interfere threads. In order to get the service of the operating system (OS), threads need to invoke system calls frequently during their lifetime. To access OS's address space, threads have to switch to the kernel state. After the service finished, the state returns back to the user state, which is the address space of threads originally. For a simple system call, kernel only uses a small part of a page, while it has to complete the above steps, which may lead to two additional row-buffer misses. Although OS invocations are usually short-lived, they are invoked frequently [20], which leads to the frequently switches between kernel state and user, intensifying interference. Figure 4 demonstrates row buffer misses proportion caused by OS to different benchmarks. In the figure, we run different benchmarks to count the misses proportion caused by invoking system calls and other guest OS interference. This figure has clearly shown OS contributes most row buffer misses to threads; the other is that interference among parallel running threads. Threads running concurrently contend shared memory in HMP systems. Therefore, memory streams of different threads are interleaved and interfere with each other at DRAM memory and virtual memory address space respectively. The results of figure 5 have proven the interference among threads. Moreover, as the threads number increased, from one thread to 16 threads, the row buffer miss rate decreases seriously. Interference among threads needs to alleviate for memory performance improvement.

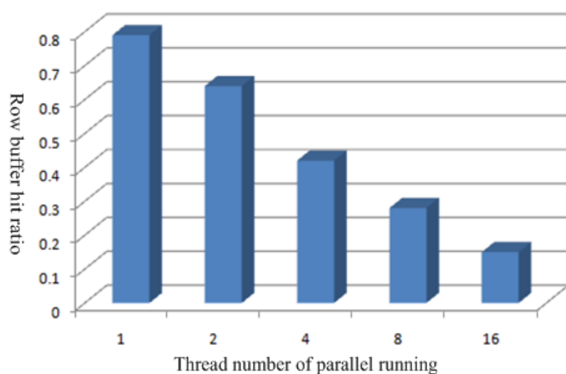


FIGURE 3. Row buffer hit ratio decreases significantly with the thread number increased.

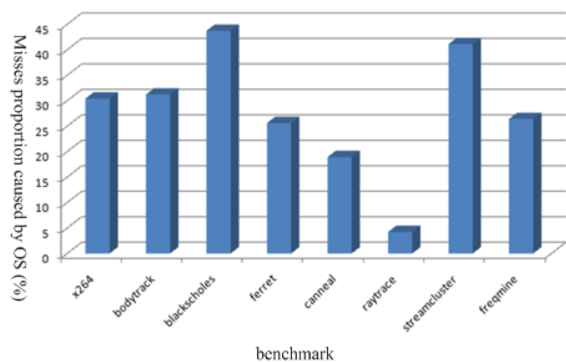


FIGURE 4. Row buffer misses proportion caused by OS.

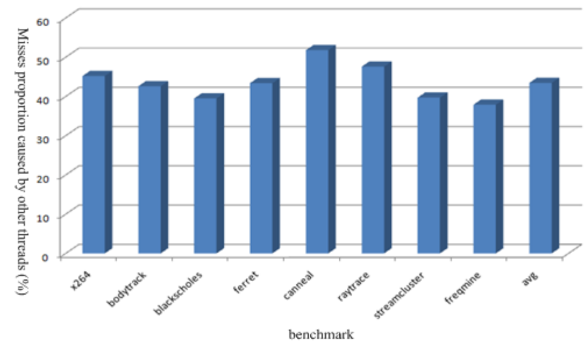


FIGURE 5. Row buffer misses proportion caused by other threads.

3) UNFAIRNESS

A good HMP system needs to guarantee that no single thread, especially the thread running on fast core, should be disproportionately slowed down. Unfortunately, threads running on current HMP system are disproportionately slowed down normalized to running alone, especially the thread running on fast core. Figure 6 demonstrates the decreased performance of each thread after running on 4-core HMP system normalized to running alone. From the figure, it's clear to see threads of parallel running decrease their performance disproportionately, moreover, the thread's performance running on fast core is slowed down more seriously, which is unbearable. In the figure, each workload, mix0, mix1 and mix2, has four threads running parallel. The first thread of the workload runs on the fast core for its more cpu-bound, and the other three threads run on the slow core. The reason of the thread running on fast core having worse performance is mainly two aspects: the

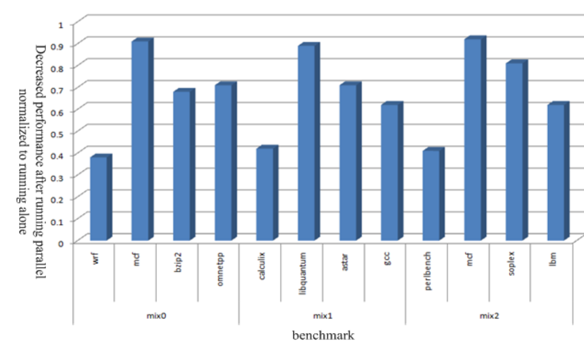


FIGURE 6. The normalized performance of each thread after running on 4-core HMP system.

first one is that the fast core thread has less memory requests for more cpu-bound, so the respond time from memory is longer. Because the mainly memory scheduling policies are FCFS-like (first come first service); the other is that the fast core is better in CPU frequency, so the CPU waiting cycles for memory respond is much more. Therefore, there is much potentiality in fairness for HMP system through shared resources management.

C. ANALYSIS OF THESE CHALLENGES

All the above three challenges are mainly from two reasons:

1) SHARE

The most important reason is sharing the resources. All parallel running threads share cache, memory, bandwidth, and so on. Therefore, every thread requests shard resources will interfere other threads, which is the root of above three challenges. If we want to address all the three challenges, we need to prevent sharing. However, in the multi-core, the hardware is expensive for every core having private all resources, and also is waste. So, the core problem is how to prevent sharing among parallel running threads in the shared environment.

2) UNLIMITED

The second reason is unlimited using shared resources from every thread. Therefore, a thread of stream application will occupy most of the shared resources, which will decrease all other threads. So, we can limit every thread occupying the amount of shared resources according to its requirement, no single will be disproportionately slowed down. But the core problem is how to estimate every thread's requirement for the shared resources. Moreover, it's not just determined by the characteristic of the thread, but also by the core performance the thread running on.

D. OUR INSIGHTS BASE ON THE ANALYSIS

1) LIMITED MEMORY BANK REQUIREMENT

Although memory bank can improve parallelism, the necessary amount of memory banks one thread requires is limited. Usually, a thread is unable to generate enough concurrent memory requests due to the combination of many factors such as memory dependency, high cache hit rate and limited number of MSHRs [21]. However, most modern systems always spread all threads' memory banks across the whole memory for accessing all banks to take advantages of bank-level parallelism, which is only excess and suffering memory interference. Therefore, we can prevent sharing among parallel running threads in the shared environment through partitioning memory banks for each thread, which eliminates inter-thread bank conflicts. Moreover, as the technique of OS page-coloring is well-known for cache partitioning [22], the technique can also be used to partition memory banks. So we can partition both cache and memory banks simultaneously adopting page-coloring technique to prevent sharing among parallel running threads in the shared environment.

2) THREAD'S REQUIREMENT FOR SHARED RESOURCES

In order to estimate the shared resources requirements for every thread, we need to analyze the characteristic of both thread and core. Firstly, we define a thread's characteristic using three components: memory intensity [23], row-buffer locality [24], and bank level parallelism [25]. Memory intensity is the frequency that a thread generates memory requests or misses in the last-level cache. Row-buffer locality is the locality that a thread hits the row-buffer. Bank-level parallelism is parallelism that a thread independent accesses to different banks in parallel. Secondly, we define a core's characteristic using three components: CPU frequency, cache size, and out-of-order window size. Therefore, we use the six components to estimate the shared resources requirements for every thread. Based on the estimation, we can allocate resources to prevent unfairness.

III. DYNAMIC RESOURCE PARTITIONING (DRP)

A. OVERVIEW OF DRP

To maximize the effectiveness of the HMP system, we propose Dynamic Resource Partitioning (DRP). It contains four parts of the DRP. First, to estimate threads' needs for shared resources, mainly shared caches, memory banks and memory bandwidth, thread's characteristic profiling is to profile threads' three components, memory intensity, row-buffer locality and bank level parallelism. Second, to estimate cores' needs for shared resources in HMP, core's characteristic profiling is to profile cores' three components, CPU frequency, cache size, and out-of-order window size. Through the six components, we can estimate the shared resources requirements for every thread running on different core. Third, to take advantages of heterogeneous performance cores, based on the estimation of both thread and core characteristic, dynamic partition shared resources to prevent interfering to realize accelerating sequential CPU-bound threads using fast cores, and improving throughput of parallel memory-bound threads using slow cores. Final, to improve fairness, we combine memory scheduling police into our DRP, which can improve throughput and fairness simultaneously to maximize effectiveness of the HMP. Figure 7 demonstrates our DRP framework.

B. PROFILING THREAD'S CHARACTERISTIC

We define a thread's characteristic using three components: memory intensity, row-buffer locality and bank level parallelism.

Memory intensity, the frequency of a thread misses in the last-level cache or generates memory requests, is used to distinguish between low memory intensity and high memory intensity. In this paper, we use last-level cache misses per thousand instructions (MPKI) to represent memory intensity. The more MPKI, the thread is higher in memory intensity. Threads with low memory intensity show almost no performance changes after allocating more memory banks. The interval time between two requests from these threads

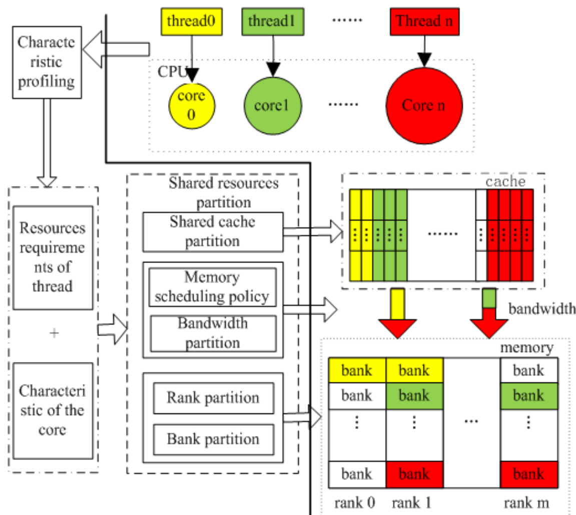


FIGURE 7. The framework of DRP.

is long for seldom generating memory requests. Therefore, bank level parallelism is useless. However, threads with low memory intensity are more sensitive to the respond time, which will reduce serious performance if waiting times is longer. In order to grantee fairness and improve system performance, we should give priority of memory scheduling for these threads. Threads with high memory intensity will have two mainly characteristics, row buffer locality and bank level parallelism. High memory intensity threads with low row buffer locality are sensitive to the number of banks. Memory banks can hide memory access latency to improve performance through bank level parallelism for these threads. But high memory intensity threads with high row buffer locality are more possible to interfere threads of parallel running. So, partition unique memory for these threads is effective.

Row buffer locality, the hit ratio of a thread accesses the row buffer, which provides temporary data storage of a DRAM row. We use average row buffer hit rate (RBH) to represent row-buffer locality in this paper. The necessary of the required bank level parallelism for one thread is limited. For low memory intensity threads, 4 or 8, even less is enough; for high memory intensity threads with low row buffer locality, 16 or 32 is enough; and for high memory intensity threads with high row buffer locality, mostly 16 is enough. More than necessary, the performance hardly improved for all threads. Therefore, we can allocate memory banks according to threads' characteristic.

Bank level parallelism, the parallelism of a thread accesses multiple and independent memory banks, which overlaps long memory access latency and improves DRAM throughput. Based on both memory intensity and row buffer locality, we determine the bank number for each thread to exploit its bank level parallelism.

C. PROFILING CORE'S CHARACTERISTIC

We define a core's characteristic using three components: CPU frequency, cache size, and out-of-order window size.

CPU frequency, the operating frequency of CPU's core, is one of most important metrics for the processor performance. The higher the frequency is for a given CPU family, the faster the processor is.

Cache size, mainly the private cache size for the core, is actually to the size of the data store, where the memory elements are actually stored. The more cache the system has, the more likely it is to register a hit on a memory access, because fewer memory locations are forced to share the same cache line.

Out-of-order window size, refers to the set of instructions which can execute out-of-order in an out-of-order speculative CPU. The instruction window has a finite size, and new instructions can enter the window only when other instructions leave the window. Instructions enter and leave the instruction window in program order, and an instruction can only leave the window when it is the oldest instruction in the window and it has been completed. Hence, the instruction window can be seen as a sliding window in which the instructions can become out-of-order. All execution within the window is speculative until it is committed.

D. PROCESS OF THE DYNAMIC CLOUD RESOURCE PARTITIONING

In this paper, we mainly partition three shared resources: memory banks, shared cache and memory bandwidth.

1) MEMORY BANK PARTITION

We partition all parallel running threads into three types, low memory intensive thread type, high memory intensive with low row buffer locality thread type, and high memory intensive with high row buffer locality thread type respectively, which accords to the five metrics, MPKI and RBH from thread, CPU frequency, cache size, and out-of-order window size from core.

In the symmetric multi-core processors (SMPs), based on MPKI and RBH, threads can be partitioned into three types. But in the HMPs, the different core characteristics the thread running on will influence the thread's characteristic. Therefore, we combine the thread's characteristic with core's characteristics. In this paper, we assume n threads are running parallel. MPKI_i and RBH_i represent the thread_i's characteristic, $0 < i < n$. We use F_i, C_i, W_i to represent CPU frequency, cache size, and out-of-order window size of the core i. Define two parameters of T_1 and $T_2, 0 < T_1, T_2 < 1$, and $T_1 + T_2 = 1$. In order to quantize the influence of the core's characteristic, use $T_1{*}F_i + T_2{*}C_i$ as the memory intensive factor, and use W_i as the row buffer locality factor. After combining core's characteristic, the thread's memory intensive characteristic stands MPKI_i by $\mathrm{MPKI}_i{*}(T_1{*}F_i + T_2{*}C_i)/\Sigma(T_1{*}F_k + T_2^{*}C_k)$ and row buffer locality characteristic stands RBH_i by $\mathrm{RBH}_i{*}W_i/\Sigma W_k$.

We define two threshold parameters MPKI_t and RBH_t for classification types. If the combined MPKI $< \mathrm{MPKI}_t$, the thread is partitioned into low memory intensive type; else if combined RBH $< \mathrm{RBH}_t$, the thread is partitioned into high

Algorithm 1 Our Memory Bank Partition Algorithm

Definition:
N: the total cores in the processor
M: the total memory banks in the server
NL: the number of low memory intensive threads
NHL: the number of high memory intensive with low row buffer locality threads
NHH: the number of high memory intensive with high row buffer locality threads
N: the total cores in the cloud computing
Our memory bank partition:
If NL==N
    Allocate M/N banks for every thread;
Else if NL==0
    Allocate M/(NHL+NHH) banks for every threads in high memory intensive with high row buffer locality type;
    If M/(NHL+NHH) <16
        Each two threads in high memory intensive with low row buffer locality type sharing M/(NHL+NHH)*2 banks;
    Else
        Allocate M/(NHL+NHH) banks for every thread in high memory intensive with low row buffer locality type;
Else
    Allocate M/(NHL+NHH) banks for every thread in high memory intensive with high row buffer locality type;
    Threads in low memory intensive type can access all banks allocated for high memory intensive with high row buffer locality threads;
    If M/(NHL+NHH)<16
        Each two threads in high memory intensive with low row buffer locality type sharing M/(NHL+NHH)*2 banks;
    Else
        Allocate M/(NHL+NHH) banks for every thread in high memory intensive with low row buffer locality type;

memory intensive with low row buffer locality type; else, the thread is partitioned into high memory intensive with high row buffer locality type.

Based on the classified types, partition memory banks to reduce interference. So, in this paper, allocate unique 16 memory banks for each thread in high memory intensive threads with high row buffer locality group; allocates one 16 banks group for some threads in high memory intensive threads with low row buffer locality group sharing; allocate all banks allocated for high memory intensive threads with low row buffer locality group to low memory intensity group. Algorithm 1 shows the pseudo code of our memory bank partition.

2) SHARED CACHE PARTITIONING

Figure 8 shows the address mapping policy of the 32 bits system. A physical address contains several common bits, which denote both OS page index and bank index, so these bits are referred to as bank color bits. In this figure, there

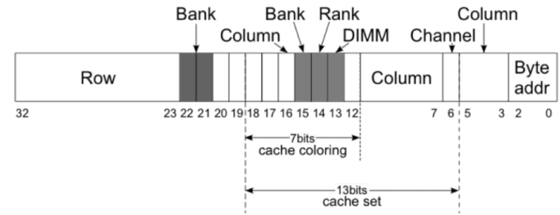


FIGURE 8. Address mapping policy.

are 5 bank color bits, bit 13 15 and 21 22, which contains 32 colors and each color represents 1 memory bank. Each OS page index (bit 12 31) also contains 5 bank color bits which designate one color, thus each physical page belongs to one bank color.

Moreover, as a shared 8MB 16-way associative last level cache, the bit 6 18 are used to choose cache set. So the bit 12 18 are cache coloring. Therefore, the shared bit 13 15 are used for both bank coloring and cache coloring. The shared last level cache is partitioned into 8 groups. Through this way, we can partition memory banks and last level cache simultaneously.

3) MEMORY BANDWIDTH PARTITIONING

After partitioning both memory banks and last level cache for all parallel running threads, the mainly contention is the bandwidth. If the bandwidth allocates to each parallel running thread fairly, the unfairness problem can be solved.

There are N parallel running VMs in the system, using $T_1, T_2, \ldots, T_N$ represent each thread. And through performance management unit (PMU), which is popular in current processors. And use the PMU, we can count committed instructions and access memory numbers/last level cache misses of each thread, which represented by IN_i and M_i respectively for thread i. B stands for the total bandwidth. Therefore, we need to determine $B_1, B_2, \ldots, B_N$ for each thread in order to fairly share.

If a thread belonged to low memory intensive type, it can use bandwidth unlimitedly for its performance. Else threads belonged to high memory intensive types, they need to use bandwidth limitedly for fairly sharing.

Therefore, the performance degradation of each thread belonged to high memory intensive types is $(IN_i + B_i)/(IN_i + M_i)$, where B_i is smaller than M_i for contending bandwidth.

So, for random i and j which $1 < i, j < N$ and both T_i and T_j belonged to high memory intensive types. They must be

$$(IN_i + B_i)/(IN_i + M_i) = (IN_j + B_j)/(IN_j + M_j) \quad (1)$$

$$(B_1 + B_2 + \ldots + B_N) = B \quad (2)$$

Through (1) and (2), we can get each B_i, which allocates for each thread. After that, the process of bandwidth partition finishes.

4) INTEGRATE MEMORY SCHEDULING INTO DRP

Although shared resources partition can reduce interference and improve fairness, it can't maximize the performance and fairness for unable to speedup the low memory intensive thread. Therefore, in order to maximize the effectiveness of the HMP system, integrate memory scheduling into our DRP.

In our memory scheduling, priority schedule memory accessing from low memory intensive threads, the Figure 9 demonstrates the example. Priority schedule memory access from fast core, which can speed up that thread. If there are more than 1 threads of low memory intensive type, priority schedule row buffer hit memory accessing. Memory accesses from high memory intensive threads are scheduled based on both bandwidth partition and row buffer hit.

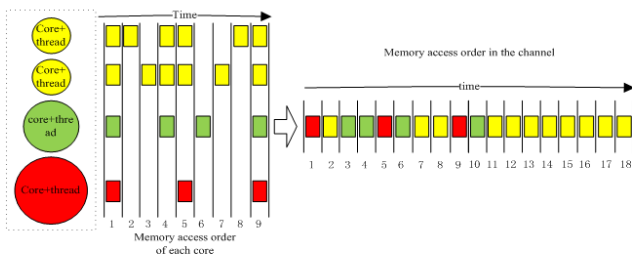


FIGURE 9. The example of the low memory intensive thread first memory scheduling policy.

IV. EXPERIMENTAL SETUP

We use MARSSX86 [26] as the base full-system architectural simulator to run Linux 2.6.31 and extend its memory part with DRAMSim simulator to simulate DDRx DRAM systems in the details. Table 1 shows the major simulation parameters of the quad core with one memory controller, one is the fast and powerful core for low memory intensive threads, three others are power efficiency cores for high memory intensive threads which one is for high row buffer locality and the other two for low row buffer locality, and most parameters are the same of the 8-core.

TABLE 1. Processor and memory configurations.

Feature	Value
CPU cores	eight cores with different erformance, 4 cores are fast, 4 cores are slow
L1 I/D cache (per core)	16KB, 2-way for fast core8KB, 2-way for slow core
L2 cache (shared)	8MB, 16-way
Cache block size	64bytes
Memory configuration	4 GB, 2 channels, 8 ranks, 8banks per rank

In order to evaluate our DRP, we simultaneously run different combinations of selected from sysbench [27], SPEC2000 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 SPEC2000 and SPEC2006 workload, it is the number of copies of the application with the name of appname. After conducting experiment to get each benchmark's memory access characteristic, we classify the benchmarks into different categories: memory-non-intensive, memory-intensive with low row buffer locality and memory-intensive with high row buffer locality.

TABLE 2. Workload description.

mix	Sysbench, SPEC2000 and SPEC2006
mix1	2-sysbench cpu, povray, tonto, calculix, perlbench, namd, wrf
mix2	2-sysbench cpu, perlbench, namd, wrf, dealII, gcc, sjeng
mix3	2-sysbench cpu, dealII, gcc, sjeng, gobmk, gromacs, h264ref
mix4	1-sysbench cpu, 1-sysbench memory, gobmk, gromacs, h264ref, bzip2, hmmer, astar
mix5	1-sysbench cpu, 1-sysbench memory, h264ref, bizp2, hmmer, astar, cactus, omnetpp
mix6	1-sysbench cpu, 1-sysbench memory, hmmer, astar, cactus, omnetpp, xalanc, sphinx3
mix7	2-sysbench memory, cactus, omnetpp, xalancbmk, sphinx3, gems, lbm
mix8	2-sysbench memory, xalancbmk, sphinx3, gems, lbm, soplex, leslie3d
mix9	2-sysbench memory, gems, lbm, soplex, leslie3d, libquantum, mcf

Evaluation Metrics: We measure system throughput using weighted speedup and fairness using maximum slowdown. *weighted_speedup* and fairness using *maximum_slowdown*.

$$weighted_speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}} \tag{3}$$

$$maximum_slowdown = max_i \frac{IPC_i^{alone}}{IPC_i^{shared}}. \tag{4}$$

V. EXPERIMENTAL RESULTS

A. PERFORMANCE ANALYSIS

We evaluate the impact of our DRP on system throughput using weighted speedup. Figure 10 provides insight into where the performance benefits of DRP and DRP-MS come from. DRP-MS represents the DRP integrated with our memory scheduling. In the figure, DBP represents the method of dynamic bank partition, TCM is one of the best previous memory scheduling method for trading off between performance and fairness in CMP, FCFS is the default memory management method of first come first service.

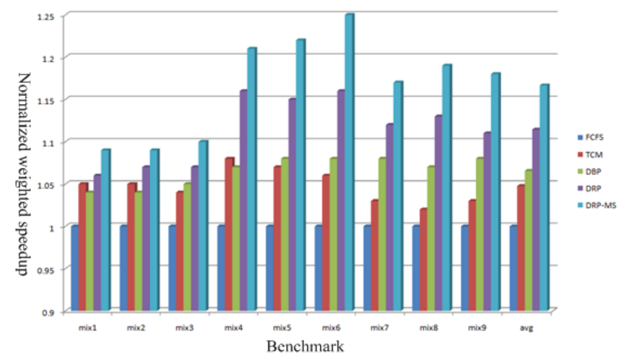


FIGURE 10. System throughput.

As expected, both our DRP and DRP-MS outperform previous methods by both speeding up the threads running on fast

core and increasing fairness. In our workload, we partition 9 mixes into three groups, the group 1 of mix1 to mix3 is low memory intensive, which mostly contains only low memory intensive benchmarks, the group 2 of mix4 to mix6 is middle memory intensive, which contains both low and high memory intensive benchmarks, and the group 3 of mix7 to mix9 is high memory intensive, which mostly contains only high memory intensive benchmarks. In the figure, we can see the group 2 has the most improvement for all the methods, this is because our experiment is based on the thread scheduling of sequential CPU-bound threads using fast cores. Therefore, all the methods behave the most effective in this group for speeding up the sequential CPU-bound threads on fast cores, which is through reducing interference. However, our DRP and DRP-MS behave better than other methods obviously. This performance improvement is mainly from speeding up the sequential CPU-bound threads on fast cores further and increasing fairness among parallel running threads.

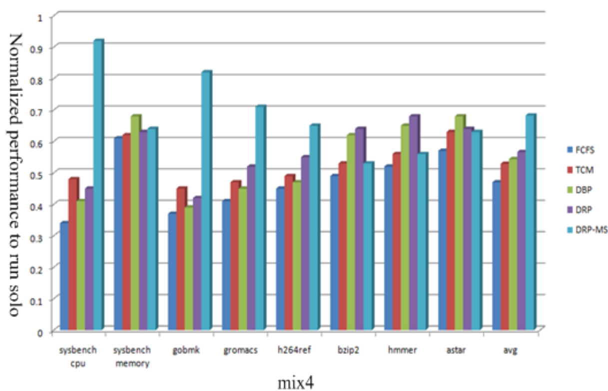


FIGURE 11. **The normalized performance of each thread in mix4 to run solo.**

Figure 11 provides the normalized performance of each thread in mix4 to run solo. In the mix4, there are 8 benchmarks, which have different memory access behavior. We place the more memory intensive benchmarks, sysbench memory, gobmk, gromacs, h264ref, on the slow core, and other 4 benchmarks on the fast core. From the figure, we can see our DRP and DRP-MS are better than others, especially the DRP-MS. This is mainly because our DRP and DRP-MS can profile both the thread and core characteristic. But other methods only can profile threads' characteristic, and the thread runs on different cores, its behavior will change. The highly improvement of DRP-MS is also from the priority schedule memory accessing from low memory intensive threads, therefore, the improvement of threads running on fast core are high, and the performance of threads running on slow core are not as well as other methods, but is almost the same.

B. FAIRNESS ANALYSIS

Figure 12 demonstrates the fairness evaluation, which is measured by maximum slowdown. DRP and DRP-MS are better than others for speeding up the threads running on fast cores.

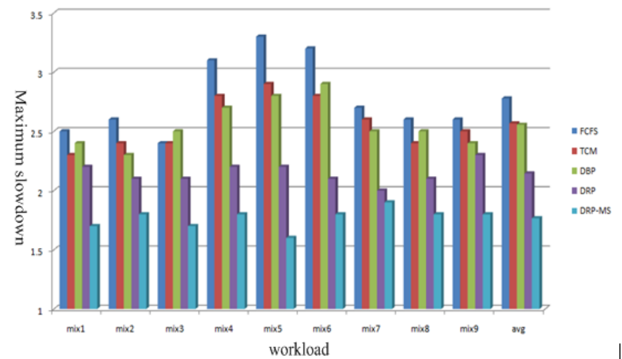


FIGURE 12. **Maximum slowdown.**

The maximum slowdown of other methods is derived from the threads running on fast core, because their performance decreases more seriously. DRP and DRP-MS can speed up the performance of threads running on fast core without decreasing other threads' performance seriously, therefore, we can improve fairness.

Moreover, both our shared cache partition policy and memory bandwidth partition policy contribute the fairness improvement. Figure 13 provides the shared cache hit ratio of each thread in mix4. The shared cache hit ratio can be improved especially for threads running on fast cores, which can contribute both performance and fairness improvement.

For our memory bandwidth partition policy, we can limit the memory access from memory intensive threads, which can meet the request from threads running on fast core firstly. This policy also can speed up the threads running on fast cores, which improves both performance and fairness.

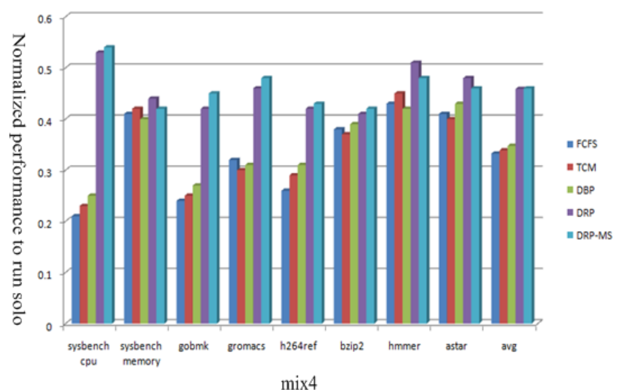


FIGURE 13. **The shared cache hit ratio of each thread in mix4.**

C. OVERHEAD ANALYSIS

Software Support. There are four parts which requires system software support, shared cache partition, memory scheduling policy, bandwidth partition and bank/rank partition. The four parts integrate the profiling thread's characteristic, profiling core's characteristic, memory bank partition algorithm, shared cache partitioning algorithm, memory bandwidth partitioning algorithm. All these processes need addition computing.

Performance Overhead. In order to evaluate the performance overhead after adopting our DRP, we compare our DRP with the default method, and the experimental results show the overhead of our DRP is negligible, below 2.5%. The result has shown our DRP behave well.

VI. CONCLUSION

In the future, the cloud computing will be the infrastructure of the smart cities to cater more and better services while cutting down the budget. So the cloud computing adopts the better single-ISA heterogeneous multi-core. However, it needs properly management to leverage the effectiveness of the HMP. In this paper, we propose a dynamic resource partitioning (DRP) for single-ISA heterogeneous multi-core, which partitions shared resources according to both threads' requirements for shared resources and the performance of their running cores. The key principle is to profile both threads' resource characteristics at run-time and cores' performance the threads running on to estimate demands for resources. Then, use the estimation to direct our resource partitioning. Moreover, we integrate our dynamic resource partitioning with current memory scheduling policies to improve system performance further for the two methods being orthogonal. Experimental results show that the proposed DRP improves system performance by 5.6% and improves system fairness by 31% over memory partitioning for symmetric multi-core. Compared to memory scheduling policies, our integrated policies behaves well both in performance and fairness. We conclude that our methods are effective in improving both system throughput and fairness for single-ISA heterogeneous multi-core.

REFERENCES

[1] J. Held, J. Bautista, and S. Koehl, "From a few cores to many: A tera-scale computing research review," Intel, Mountain View, CA, USA, White Paper, 2006. [Online]. Available: http://download.intel.com/research/platform/terascale/

[2] R. Rodrigues, A. Annamalai, I. Koren, and S. Kundu, "Scalable thread scheduling in asymmetric multicores for power efficiency," in *Proc. IEEE 24th Int. Symp. Comput. Archit. High Perform. Comput.*, Oct. 2012, pp. 59–66.

[3] R. Kumar, D. M. Tullsen, and N. P. Jouppi, "Core architecture optimization for heterogeneous chip multiprocessors," in *Proc. 15th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2006, pp. 23–32.

[4] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008.

[5] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2010, pp. 29–40.

[6] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating Amdahl's law through EPI throttling," in *Proc. ISCA*, 2005, pp. 298–309.

[7] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Proc. MICRO*, vol. 36. 2003, p. 81.

[8] G. Jia, G. Han, J. Jiang, and J. J. P. C. Rodrigues, "PARS: A scheduling of periodically active rank to optimize power efficiency for main memory," *J. Netw. Comput. Appl.*, vol. 58, pp. 327–336, Dec. 2015.

[9] T. Y. Morad, U. C. Weiser, and A. Kolodny, "ACCMP: Asymmetric cluster chip multi-processing," in *Proc. TRCCIT*, 2004, pp. 1–5.

[10] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proc. MICRO*, 2010, pp. 65–76.

[11] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *Proc. ISCA*, 2008, pp. 63–74.

[12] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *Proc. HPCA*, 2012, pp. 1–12.

[13] G. Jia, G. Han, J. Jiang, and A. Li, "Dynamic time-slice scaling for addressing OS problems incurred by main memory DVFS in intelligent system," *Mobile Netw. Appl.*, vol. 20, no. 2, pp. 157–168, 2015.

[14] G. Jia, G. Han, L. Shi, J. Wan, and D. Dai, "Combine thread with memory scheduling for maximizing performance in multi-core systems," in *Proc. ICPADS*, 2014, pp. 298–305.

[15] G. Jia, L. Shi, J. Wan, Y. Yuan, X. Li, and D. Dai, "PUMA: Pseudo unified memory architecture for single-ISA heterogeneous multi-core systems," in *Proc. RTCSA*, 2014, pp. 1–10.

[16] G. Jia, X. Li, J. Wan, L. Shi, and C. Wang, "Coordinate page allocation and thread group for improving main memory power efficiency," in *Proc. HotPower*, 2013, Art. ID 7.

[17] M. Xie, D. Tong, K. Huang, and X. Cheng, "Improving system throughput and fairness simultaneously in shared memory CMP systems via dynamic bank partitioning," in *Proc. HPCA*, 2014, pp. 344–355.

[18] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, "MemScale: Active low-power modes for main memory," in *Proc. ASPLOS*, 2011, pp. 225–238.

[19] S. Cho and L. Jin, "Managing distributed, shared L2 caches through OS-level page allocation," in *Proc. MICRO*, 2006, pp. 455–468.

[20] T. Li, L. K. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio, "Understanding and improving operating system effects in control flow prediction," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 68–80, 2002.

[21] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *Proc. PACT*, 2012, pp. 367–376.

[22] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *Proc. IEEE 14th HPCA*, Feb. 2008, pp. 367–378.

[23] M. Xie, D. Tong, Y. Feng, K. Huang, and X. Cheng, "Page policy control with memory partitioning for DRAM performance and power efficiency," in *Proc. ISLPED*, 2013, pp. 298–303.

[24] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Proc. MICRO*, 2011, pp. 374–385.

[25] G. Jia, G. Han, A. Li, and J. Lloret, "Coordinate channel-aware page mapping policy and memory scheduling for reducing memory interference among multimedia applications," *IEEE Syst. J.*, 2015, doi: 10.1109/JSYST.2015.2430522.

[26] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A full system simulator for x86 CPUs," in *Proc. DAC*, 2011, pp. 1–6.

[27] A. Kopytov. (2004). *SysBench: A System Performance Benchmark*. [Online]. Available: http://sysbench.sourceforge.net/index.html

[28] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *Proc. 15th ASPLOS*, 2010, pp. 335–346.

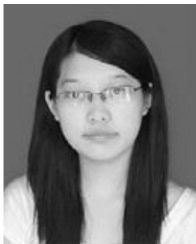
[29] Y. Zhang, S. He, and J. Chen, "Data gathering optimization by dynamic sensing and routing in rechargeable sensor networks," *IEEE/ACM Trans. Netw.*, to be published.

[30] S. He, J. Chen, F. Jiang, D. K. Y. Yau, G. Xing, and Y. Sun, "Energy provisioning in wireless rechargeable sensor networks," *IEEE Trans. Mobile Comput.*, vol. 12, no. 10, pp. 1931–1942, Oct. 2013.

GANGYONG JIA (M'13) received the Ph.D. degree from the Department of Computer Science, University of Science and Technology of China, Hefei, China, in 2013. He is currently an Assistant Professor with the Department of Computer Science, Hangzhou Dianzi University, China. He has authored over 30 papers in related international conferences and journals. His current research interests are power management, operating system, cache optimization, and memory management. He has served as a Reviewer of Microprocessors and Microsystems.

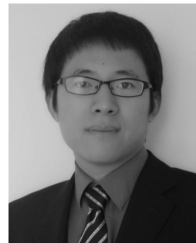
GUANGJIE HAN (S'03–M'05) received the Ph.D. degree from Northeastern University, Shenyang, China, in 2004. From 2004 to 2006, he was a Product Manager with ZTE Company. In 2008, he finished his work as a Post-Doctoral Researcher with the Department of Computer Science, Chonnam National University, Gwangju, Korea. From 2010 to 2011, he was a Visiting Research Scholar with Osaka University, Suita, Japan. He is currently a Professor with the Department of Information and Communication System, Hohai University, Nanjing, China. He has authored over 180 papers in related international conference proceedings and journals, and holds 55 patents. His current research interests include sensor networks, computer communications, mobile cloud computing, and multimedia communication and security.

Dr. Han is a member of the Association for Computing Machinery. He received the ComManTel 2014, ComComAP 2014, and Chinacom 2014 Best Paper Awards. He has served as a Co-Chair of over 20 international conferences/workshops and a Technical Program Committee Member of over 70 conferences. He has served on the Editorial Boards of up to 16 international journals, including the *International Journal of Ad Hoc and Ubiquitous Computing*, the *Journal of Internet Technology*, and *KSII Transactions on Internet and Information Systems*. He has served as a Reviewer of more than 50 journals.



JINFANG JIANG received the B.S. degree in information and communication engineering from Hohai University, China, in 2009. She is currently pursuing the Ph.D. degree with the Department of Information and Communication System, Hohai University, China. Her current research interests are security and localization for sensor networks.

NING SUN received the Ph.D. degree in computer science from The Chungbuk National University, Korea, in 2013. Since then, she has been a Lecturer with the Department of IOT Engineering, Hohai University, China. She is currently investigating wireless sensor networks, Internet of Things, and network security. Her research interests are in the design and evaluation of network architectures and protocols.



KUN WANG (M'12) received the Ph.D. degree from the School of Computer, Nanjing University of Posts and Telecommunications, China, in 2009. He is currently a Post-Doctoral Fellow with the Electrical Engineering Department, University of California at Los Angeles. He is also an Associate Professor with the School of Internet of Things, Nanjing University of Posts and Telecommunications. He has published over 50 papers in related international conferences and journals, including the *IEEE Communications Magazine*, the IEEE GLOBECOM 2013, and the IEEE ICC 2014. His current research interests include wireless sensor networks, delay-tolerant networks, stream computing, ubiquitous computing, mobile cloud computing, and information security technologies.