A task migration algorithm for power management on heterogeneous multicore

Manman Peng[1, a], Wen Luo[1, b]

[1]School of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China.

[a]pmmbysj@hnu.cn, [b]linuxc_lw@163.com

Keywords: cache contention, power consumption, heterogeneous multicore system, cache miss.

Abstract. The advantages of the increased chip integration for multicore technology exponentially improve the performance of the processor chip. However, with the increased number of cores, the power consumption of the system has become an increasingly important problem. Compared with homogeneous multicore, the main advantage of heterogeneous architecture on the power consumption is that it allows for the inclusion of processors whose design is specialized for particular types of jobs, and for jobs to be assigned to a processor best suited for that job. In most multicore platforms, cache resources is a scarce resource, different cores share on chip caches. Without effective management by the scheduler, such caches can cause thrashing that severely degrades system performance. An excellent task scheduling algorithm is an effective method to reduce Cache contention. The purpose of this paper is through the program feature classification reduce process cache contention, let the process which can better make use of the performance of processor execute upon high frequency cores, improving the processor performance reduce the execution time to achieve the purpose of reducing power consumption in a heterogeneous multicore system.

Introduction

In recent years, the improvement of processor performance and quantity leads to the rapid increase of energy consumption, which is limited by increasing the number of cores to improve the performance of processors. Multicore increased the energy consumption of the chip and the burden of cooling. The increase of the core increases the energy consumption of the chip and the cooling load. At the same time, the growth of energy consumption will produce more heat and lead to the decrease of CPU life, and the reliability of the system is reduced. Due to the limitation of semiconductor process, physical characteristic and thermal diffusion technology, processor speed is not expected to have a significant raise in the near future. In order to further improve the processor capacity, some hardware technology, such as closing an idle circuit and reducing the energy level of functional units which not being fully utilized [1, 2], to make the processor layer of energy management and reduce power consumption has become a reality [3]. Dynamic voltage scaling and dynamic frequency scaling is a kind of effective power optimization techniques [4, 5], DVFS is according to the varied scenarios to be run with different frequency and voltage settings to meet the current of the circuit timing and performance requirements. But the optimal frequency and voltage of the multi-core processor are not suitable for all the cores on the processor. The global frequency is adjusted to make the processor run at the same frequency, but the different optimal operating frequency of different kernel tasks makes it difficult to match the global frequency.

A chip multiprocessor (CMP) consists of several cores which can execute tasks independently. Due to the budget and chip area limit, last level cache is usually shared among cores. If tasks running on different cores access the shared cache intensively and concurrently, it may lead to high cache miss rate and significant performance degradation. A commonly-used method is to co-schedule a task with good anti-interference ability and a task with poor anti-interference. However, if tasks have similar anti-interference abilities, it becomes difficult to generate a proper task assignment. We analyze the cache loss, cache contention, task IPC, and other features on the

task of running on multi-core heterogeneous processors. Although multi core processors can operate at different frequencies, it is still possible to be considered homogeneous processors. We analyze the characteristics of tasks that run on multiple processors considering cache contention. With the scheduler of Linux, scheduler does not consider the cache contention between processes running on the same processor [6, 7]. The cache conflict may be very serious and cause the decline of the performance. In addition, if there are two tasks, the previous one is scheduled to run on the low frequency core, the latter one is scheduled to run on the high frequency core. So the two tasks will also lead to the use of system resources unreasonable to reduce the performance of the system. In view of the above situation, we proposed scheduling strategy. Depending on the task's sensitivity to cache, the tasks are classified into three categories, the category takes into account of two aspects. In later chapter, this paper will detail. The tasks are assigned to different processors, in order to reduce the inter task cache contention. Then in each category, threads conducting CPU intensive tasks may execute upon high frequency cores, while threads conducting memory intensive tasks may execute upon low frequency cores, which consume significantly less power and shorten the total execution time.

Background

Cache features of quad core processor. In a lot of recent quad core processors, cache is divided into two halves, each half being shared by two adjacent cores. Most commercial multi-core processors still use a single processor cache design, but in the original single core processor does not take into account the interference between. Due to the adjacent core shared cache, once two processes that have a great need for cache run on two adjacent core at the same time, there is no effective management scheduler, this kind of situation may lead to cache thrashing, it will seriously degrade the performance of the system, make the task execution time stretched, consume more power consumption.

Scheduling algorithm is not mature. Until the present stage, the research on the algorithm of CMP architecture system is not mature in the world. Although Windows and Linux operating system can be used as a CMP multi-core architecture of the operating system, task scheduling algorithm of the operating system support for multi-core but not direct support for heterogeneous multi-core architecture, in fact, these algorithms are for SMP (single chip architecture). Most of the current scheduling algorithms are based on homogeneous multi-core processors, but we in multiprogramming computing environment, the task execution process can reflect the different operation characteristics and different requirements of hardware resources. The Linux scheduler will distribute the process to each processor on average. Linux scheduler is fair treatment to all processors without considering the contention for shared cache. After inappropriate scheduling process may lead to serious conflict. Due to cache conflicts, frequent page replacement slows the process execution speed and increases the consumption of energy.

The hardware performance counters. The performance parameters of the task can be recorded by perf_Event API using performance counters. In today's many processor performance counter is available. Performance counters can be programmed to monitor a number of events. For example, Intel has specified a set of performance events that can be monitored by using performance counters. These events include the clock cycle, the number of instructions executed, cache miss, cache reference, and the event related parts, etc… Further, many special processors have their own special events.

Analysis

Modern processors contain multiple cores that enable them to execute multiple applications simultaneously on a single chip. All the tasks need to access the memory, all data is stored in memory, but a small part of the data from memory mapped into the cache. With the increase of number of core on the chip, the pressure on the maintenance of concurrent applications to share the cache demand is also increasing. In most tasks, with the increase in the size of the cache, the

number of cache miss decreased [8]. It can be seen that the allocation of a larger cache can significantly improve the performance of tasks to reduce the execution time. The SPEC CPU™ 2006 benchmark is SPEC's next-generation, industry-standardized, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler. This benchmark suite includes the SPECint benchmarks and the SPECfp benchmarks. The SPECint 2006 benchmark contains 12 different benchmark tests and the SPECfp 2006 benchmark contains 19 different benchmark tests. We analyze the instructions of the benchmarks in cpu2006 spec. According to our statistical results, we find that each program has its own memory access characteristics. Nearly half of the instructions are required for memory access. So the size of the cache is very important for the program. But the effect of cache size on the performance of different benchmarks is different. The benchmarks are divided into three categories by testing the execution time of the benchmarks using different cache sizes. To simplify the experiment, we ran each benchmark with both 2MB cache size and 8MB cache size, each program had at least 1GB physical memory which is larger than memory footprint of any program from SEPC CPU2006 on our experimental machine. This ensures that I/O overhead does not increase. Type A: compared to the speed of benchmark allocated 8MB cache size, the speed of benchmark allocated 2MB cache size is lower more 10%; the program is sensitive to the last level cache. Type B: compared to the speed of benchmark allocated 8MB cache size, the speed of benchmark allocated 2MB cache size is lower more 5%; Type C: compared to the speed of benchmark allocated 8MB cache size, the speed of benchmark allocated 2MB cache size is lower less 5%; The programs of type A are sensitive to LL cache, these programs want large LL cache size to improve the performance. The LL cache access rate of type B programs is high, but the need of cache size is not as large as type A. The benchmarks of type C demand for cache is the lowest. In the paper [8], the author put the threshold of cache access ratio as a classification standard cannot be a good response to the impact of cache on the performance of the program, such as libquantum and bzip2. The cache access rate of libquantum is far greater than the cache access rate of SPHINX3, but with the increase in the size of the cache, the performance of libquantum is less than 5%, and the performance of SPHINX3 is higher than 5%. The effect of cache size on performance and the impact of cache size on cache access is the opposite. In the analysis of the effects of cache size on program performance, we found that when the cache size increased from 2MB to 8MB, the MPKI (missing number per thousand instruction cache) of libquantum dropped from 13.4 to 12.7 and the MPKI of sphinx3 dropped from 9.36 to 6.34. The effect of cache size on performance is the same as the impact of cache size on MPKI. The relationship between performance and MPKI can also be applied in other benchmarks. So in the experiment of this paper, compared with the use of cache access ratio, the MPKI can better response benchmarks for the sensitivity of cache.

In addition, in heterogeneous multi-core processors, different core work at different frequencies. Tasks run on the core of high frequency compared to run on the core of the low frequency, the running time of the tasks will be reduced in different degrees.

At the time of multitasking, choose which tasks to run on the core of high frequency to reduce the running time of all tasks are of great significance. The IPC (the number of cycles per instruction) of each task may be different at different frequencies. Tasks run at the core of a high frequency and run at a low frequency, compared to the core task of the IPC will improve very significantly. As shown by table 1, we run Bzip2 at the frequency of 1.6 GHZ and 2.4 GHZ and the IPC of Bzip2 are 1.3 and 1.81 respectively. The growth of the IPC ratio is 1.39 from low frequency core to high frequency core. We run Sphinx3 at the frequency of 1.6 GHZ and 2.4 GHZ and the IPC of Bzip2 are 1.19 and 1.42 respectively. The growth of the IPC ratio is 1.19 from low frequency core to high frequency core. In order to make full use of heterogeneous multi-core performance, so we should choose the right task to run on a suitable core.

Table 1 The IPC comparing

Benchmarks	1.6GHZ	2.4GHZ	The growth of the IPC ratio
Bzip2	1.3	1.81	39%
Sphinx3	1.19	1.42	19%

Migration Algorithm

All programs are assigned to different cores, these which are allocate to the adjacent cores of the same processor share the same resources of the processor. If there is more than one task executed on the processor, these tasks compete for the shared cache; the competition affects the performance, and the execution time will increase if the competition is serious. For example, two tasks of type A are allocated to the adjacent cores of the same processor, because of serious contention for shared cache, the competition slowdowns the execution speed, the longer execution time causes more energy consumption. If one task of type A and one of type C are allocated to the same processor, the contention for shared cache is not serious, this is exactly what we want. According to the experiment result, we proposed a scheduling principle. Tasks of type A are executed with tasks of type B can get fine performance and less energy consumption. The tasks of type A cannot be executed on the same processor at the same time. The tasks of type A should be scheduled first with tasks of type C together. If there is not tasks of type A, tasks of type B should be considered prior to type C.

In heterogeneous multi-core architectures, the frequency of the core is different, and the IPC is different when the tasks are executed on the same core. In order to reduce cache resource competition, the migration algorithm put the task which is sensitive to shared cache and the task which is not sensitive to shared cache into adjacent cores, such as a task in type A and the other task in type C, and let the task which has high normal IPC in the same type run on the high frequency core.

Our goal is to combine tasks with type to reduce cache contention, let the cup-intensive tasks run on big core, ultimately reduce energy consumption. When a new task is coming, we verify the type of the task. If the type of task is verified, we utilize this characterization to schedule the task to the processor which is suitable for this task. If there is a new task and its type is verified, first, we examine the runqueues of the processors, obtain the processor load in the each runqueues to check whether the runqueues are balance. According to load balanced situation, we verify the processor id first, and further confirm the core to be allocated to the task. When the load of the runqueues are balanced. If the normal IPC of new task is higher than the normal IPC of the same type tasks of small cores, we let the new task run on the big cores. If not, we let the new task run on the small cores. When the load of the runqueues are unbalanced. If the normal IPC of new task is lower than the normal IPC of the same type tasks of big cores, we let the new task run directly on small cores. If not, we let the task which has the lowest normal IPC run on small cores and schedule the new task to the processor which is suitable for this task.

```
Algorithm; Task migration algorithm
New task P
Check the type of task P
/* tp is the type of the task P
load_p_i is the number of the program of tp on the two processor
*/
If load_p_0==load_p_1
  If IPC> p_0_IPC
        Match by the type and assigned the core of processor 1
  Else
        Match by the type and assigned the core of processor 0
        Migration the task which has the maximum normal IPC of tp from processor 0 to processor 1
Else
  If IPC<p_1_IPC
        Match by the type and assigned the core of processor 0
  Else
        Match by the type and assigned the core of processor 1
        Migration the task which has the minimum normal IPC of tp from processor 1 to processor 0
```

Experiment

Hardware and software: the machine it has two quad core Intel Xeon e5 processor and 6GB of memory, each processor includes four cores, two of which share 4MB of L2 cache. Operating system using Linux Ubuntu 10 with a version 2.6.34 kernel. In order to create a multiprocessor architecture with cores operating at different frequencies, we frequency scaled each core using the Linux “cpufreq” kernel module. We changed the default Linux CPU frequency scaling governor, which is the “on demand” governor, to the “userspace” governor, allowing us to then adjust the CPU frequencies for the individual cores. We partitioned the cores into hardware partitions using “CPUSETS for Linux”. We partitioned the cores into two cpusets, with each core in the same cpuset operating at the same frequency, but cores from different cpusets operating at different frequencies. We choose two adjacent cores from the first processor as cpuset1 and choose two adjacent cores from the first processor as cpuset2. The first cpuset contained two cores, each operating at 1600 MHz. The second cpuset contained two cores, each operating at 2400 GHz. We used the standard “perf” hardware performance counters subsystem in Linux to obtain per core and per task performance statistics. These statistics were used at runtime to give feedback to the operating system concerning an application’s execution characteristics and to gather performance data. In the test, we choose two type A benchmarks (bzip2, sphinx3), two type B benchmarks (libquantum, bwaves) and two type C benchmarks (namd, milc).

The first experiment we choose two type A benchmarks (bzip2, sphinx3) and two type B benchmarks (libquantum, bwaves). We want to get the normalized IPC to verify that the migration algorithm can improve the performance. The migration experiment, we adopt the migration algorithm running bzip2, sphinx3, libquantum and bwaves in four cores that two are adjacent. Bzip2 and libquantum are executed on two adjacent cores on the one processor, and the other two benchmarks are executed on two adjacent cores on the other processor.

Table 2 The four benchmark’s IPC

Benchmarks	IPC(scheduling)	IPC(contrast)	IPC improvement	Total
Bzip2 Bwaves	3.52	3.1	13%	5.03%
Sphinx3 Libquantum	2.54	2.66	-4.5%	

We obtain the IPC of the four benchmark programs by the hardware performance counter. In contrast tests sphinx3 and bwaves run on high frequency processor, bzip2 and libquantum run on low frequency processor. Using the migration algorithm in the experiment, bzip2 migrate to the big core, bzip2 and bwaves run on high frequency processor, the total IPC of bwaves and bzip2 was improved. sphinx3 migrate to the small core, sphinx3 and libquantum run on low frequency processor. As shown in table 2, although the IPC of sphinx3 was reduced, in general, the total IPC was improved by 5.03%. Generally speaking, there is a close relationship between IPC and performance. IPC improved, the performance also can be improved. As shown in Figure 1, we can see that all benchmark’s runtimes are shorten except sphinx3. The effect of performance improvement of bzip2 is the most obvious. The performance is improved by 5.1%. The energy consumption is decreased by 6.45% than the linux scheduling policy.

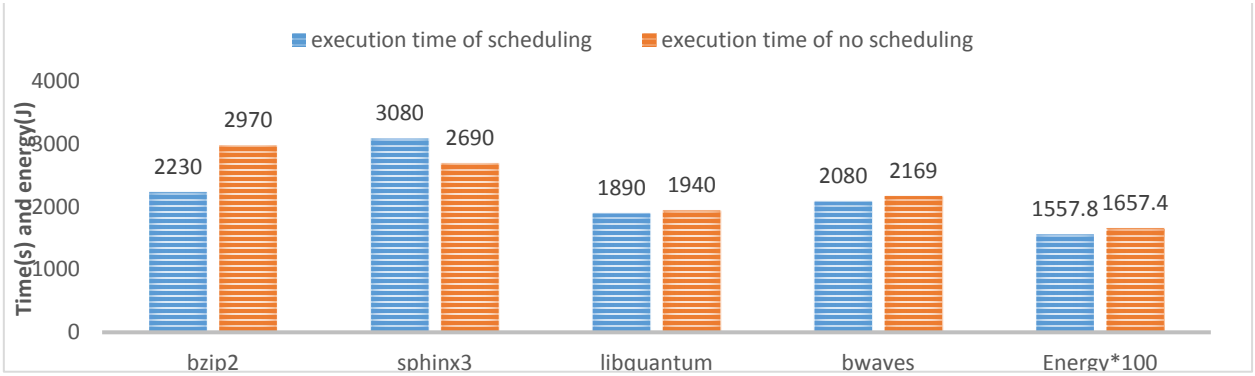

Figure 1. Energy and Runtime of Benchmarks

The same to last experiment we choose two type A benchmarks (bzip2, sphinx3) and two type C benchmarks (namd, milc). We get the result of the experiments. The runtime and energy consumption are compared to the result of no scheduling. Figure 3 shows the runtimes and energy consumption of benchmarks when scheduling is applied, relative to the runtime and energy consumption using standard Linux scheduling. As shown in Figure 2, we can see that energy consumption is less than that before scheduling and all benchmark's runtimes are shorten except namd. The performance of benchmarks is improved and the runtime is shortened. The performance is improved by 12.3%. The energy consumption is decreased by 6.5% than the Linux scheduling policy.

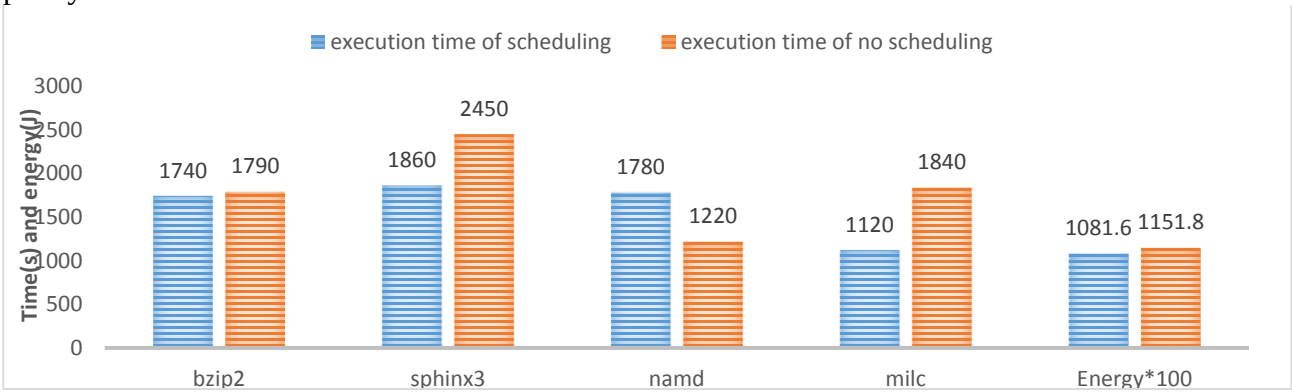

Figure 2. Energy and Runtime of Benchmarks

Summary

In this paper, the main motivation of our research was to design methods to allocate CPU resources in a multicore processor system with the goal of lowering the global power budget and creating a minimal performance loss. To accomplish this goal, we used" task-driven" feedback, in which an executing task gave feedback to system at runtime, and the system, in turn, dynamically scheduled schedule the task to the processor which is suitable for this task based upon this feedback. We have analyzed scheduling for avoiding resource contention and for optimal scheduling policy. We found that the energy consumption and the performance are supplement and complement each other. Cache contention has been a serious problem. By reducing conflict, we improve the performance, and shorten the execution time, and reduce the energy consumption.

References

[1] Burd T D, Brodersen R W. Energy efficient CMOS microprocessor design[C]// Proceedings of the 28th Hawaii International Conference on System Sciences. IEEE Computer Society, 1995:288-297 vol.1.

[2] Chandrakasan A P, Sheng S, Brodersen R W. Low-power CMOS digital design[J]. IEEE Journal of Solid-State Circuits, 2010, 27(4):473-484.

[3] Gruian F. System-Level Design Methods for Low-Energy Architectures Containing Variable Voltage Processors[C]// Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers. Springer-Verlag, 2000:1--12.

[4] Venkat Rao, Nicolas Navet, Gaurav Singhal, A Kumar, GS Visweswaran. Battery aware dynamic scheduling for periodic task graphs[c]. Parallel and Distributed Processing Symposium, 2006

[5] Aydin H, Melhem R, Mossé D, et al. Power-Aware Scheduling for Periodic Real-Time Tasks[J]. Computers IEEE Transactions on, 2004, 53(5):584-600.

[6] Chu T H, Lu W W. Cache utilization-aware scheduling for multicore processors[C]// Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on. 2012:368-371.

[7] Sun Z, Wang R, Zhang L, et al. Cache-aware Scheduling for Energy Efficiency on Multi-Processors[C]// Proceedings of the 2012 International Conference on Computer Distributed Control and Intelligent Enviromental Monitoring. IEEE Computer Society, 2012:182-186.

[8] Qureshi B M, Patt Y. Utility-based partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches[C]// In International Symposium on Microarchitecture (MICRO-39. 2010.