Effect of CPU scheduling on power consumption.

Abstruct

With the development of multi-core processors and is widely used on mobile laptops, the existence of multi-core processors advantages and disadvantages is hugely magnified. Undoubtedly, multi-core processors can improve performance of the system to the extent it is on. However, the performance improvement will be accompanied by more power consumption, although for desktops, these power hardly anything, but for a large group of servers and personal mobile laptops only, which will seriously affect the power consumption laptop device's battery life and a large group of aging servers. Multicore processors are divided into homogeneous and heterogeneous processors, heterogeneous processors can control part of the core such that the frequency of the voltage change. This article are based on heterogeneous multi-core processor group.

Despite the software level issues aside, only to multi-core heterogeneous processor itself, the following problems can cause high power consumption: cache consistency, multi-core performance and temperature control. In summary, this paper will be to heterogeneous multi-core processor-based scheduling an improved algorithm to improve power consumption.

When the process is assigned to a CPU, the CPU scheduler will wait in the queue waiting for the process of scheduling. The Linux finally adopted scheduling algorithm is CFS completely fair scheduling algorithm. Although CFS's performance is good, but it is the lack of consideration in terms of power consumption. Because the so-called completely fair is actually unfair. In other words, a IO bound processes and a CPU bound process under the same set of system, the power consumption in terms of respect is not fair. So for IO bound processes to punish and CPU bound processes compensate, so as to reduce the CPU idle time to reduce operating time and reduce power consumption.

After completion of the code modify and compile the installation is successful, the hardware implementation of this article will be described in detail later. Finally, in order to have a future multi-core heterogeneous processor environment presented challenges for software designers and the corresponding solutions. It referred to herein for the scheduler improvements need to be further improved. Improve to reach the original scheduler in the true sense, at the end of this article will work in future profiling.

Key word: Heterogeneous multi-core processor, CPU scheduler, Completely fair scheduler, Power Consumption, Cache miss rate, Performance

TABLE OF CONTENTS

1 INTRODUCTION

1.1 Research's Background

1.1.1 Power consumption among Multi-cores

Booming demand for scientific and technical computing requires computing system with more powerful processing ability. With the rapid development of the transistor fabrication, Moore's Law still plays its important role and continues to promote the use and development of the computer industry in the next ten years. As early as semiconductor technology has entered nanometers generation chip integration, we can put more transistors in the same area. but, line delay problems cannot be ignored, the processor frequency is very difficult to increase more, power consumption per unit area continue to increase, designing becomes more and more knotty. These limiting factors shows that the mono-core processor cannot fully use transistors to meet the demand for performance and power consumption, so multi-core processors come.

Recent advances in processor production have led to not only an increase in the number of cores on a single chip, but also a concomitant increase in power consumption and heat dissipation associated with these high performance systems. For example, a 2.8 GHz Intel Pentium G840 2 chip consumes around 65 W, while a 2.9 GHz AMD A6-3650 4 chip of the same speed consumes around 100 W.

It is no doubt that increased power consumption will lead to an increase in heat dissipation more serious. It is a vicious circle. So it leads to higher costs for air conditioning, better thermal packaging, fans. Increased heat and temperature of hardware (not only CPU), can also lead to decreased longevity and greater incidence of system and hardware breaking down. Which is worse is that the highly integration a multicore chip make it more difficult to satisfy thermal constraints. Finally, power savings and decreased heat production can be crucial in notebook computers, where battery life is a constraint and cooling of hardware components is difficult, and in server farms, where even a slight decrease in the power consumption of each server can cause a significant savings for the entire system.

Scaling a CPU's frequency can dramatically affect its power consumption. In fact, the power dissipation at the output pins of a core is directly proportional to its frequency and is governed by the equation:

$$P = \frac{1}{2} C V^2 f$$

where f is the effective bus frequency.

CPU frequency scaling may be used to produce an architecture with cores operating at different frequencies. Heterogeneous architectures may achieve a higher performance per watt than comparable homogeneous systems due to the ability of each application to run on a core that best suits its architectural properties. For example, CPU bound process will execute upon high frequency cores, but IO bound process has to execute upon low frequency cores, which consume significantly less power.

1.1.2 Power Consumption in one specific core

After the scheduler assigning the process to a specific core, scheduler should deal with the problem that select which process that is in the ready queue of the specific core should be executed first. There are many scheduling algorithm have ever been used or are used now, for example, the most primitive are: first com first scheduled(FCFS), shortest job first(SJF). Those two never think about the user's requirement and have very bad real-time performance. Then FPF and round robin(including some other algorithms which are derived from RR, e.g. multilevel ready queue), but they are static and also have trouble in supporting real-time process and interactive process. In order to fix the problem, staircase scheduler and RSDL（The Rotating Staircase Deadline Schedule） have been raised. Their main idea is fair, so that scheduler can give some compensation to the process according to the sleep time. Finally, the CFS(Completely fair scheduler) which is combined the advantages of RSDL/SD and abandon the aspect of interactive process. Thanks to its good performance in a lot of tests, it is the state of the art.

But we should realize that the best in the average performance does not means the best of energy saving. As we know that CFS gives every process a weight which is based on its priority, the higher priority possesses the larger weight. And the process with larger weight has more possibility to be selected at first and has more time to executed. Although the algorithm carries out the idea of completely fair, if a process with higher priority but it is exactly I/O bounded process, the whole system will wait the process for a long time, which is obviously high power consumption.

So if we can do some modification of CFS in Linux, we may gain some energy saving and even performance improvement.

1.1.3 process and power consumption

IO bound processes a problem of memory bandwidth saturation. As we know that it takes time for system to move data from the main cache to other level caches and CPU. So cores have to wait until the data has arrived. Thus IO bound processes would lose some performance because of the low speed of data transformation when it compares to the relatively high speed of data access. So obviously, if there is a task which is consisted of CPU

bound processes and IO bound processes, scheduler can assign the CPU bound process to high frequency cores and assign the IO bound process to low frequency cores. The system can gain some performance and also power saving.

In Linux2.6, taking the advantages of dynamic priority and task preemption, the scheduler reduce the response time of real-time process extremely and also increase the ratio of utilize the CPU. In other words, in order to prevent from the starvation, Linux 2.6 can modify the priority of a process dynamically. The priority of a process is defined by nice values; higher nice value means lower priority. In addition, Linux 2.6 enable the preemption, which means if a low priority process is executing and a high priority process comes, the low priority process has to stop at once and let the high priority process execute. At beginning, all the processes have the same nice value 0 and task would be assign different priority based on their nice value, which would finally influence the order of executing.

Let CPU bound process migrate from low frequency CPU to high frequency CPU can benefit not only the process but also the whole system a lot, in the same taken, if scheduler move IO bound process from high frequency to low frequency, although the process's performance has been reduce, for the whole system, it can save more waiting time and power consumption. Thus One of this research's goals is to identify and quantify these advantages.

1.2 Research's Work:

This research will study the multi-core system and analysis the power consumption. Then this research also focuses on the scheduling algorithm in one specific core and come up with a modified CFS algorithm. So our paper offers the following contributions.

1.2.1 Study of multi-core System's power consumption

The First part of the report is the study of multi-core system's charactoristics and why those charactoristics would lead to more power consumption. Then the report would introduce some ways to fix the problem.

1.2.2 Scheduler inner core

The algorithm is a modified CFS algorithm. processes' weight will be calculated not only base on its priority (nice value), but also based on its cache miss rate. Because of the changed weight, time slice and the chance of being selected first will so change. And the cache miss rate will be record dynamics, so the weight will also dynamically change to let CPU bounded processes have more possibility to be inserted into the head of the ready queue and have longer execution time.

In the modified CFS, the process will call the library PAPI to calculate the cache miss rate and then use a system call to feed the data into kernel level, finally the scheduler will use the data to calculate the new time slice and virtual run time of the process. However, the cache miss rate would change the original weight, but it would not change the priority of the process, which means that the new modified CFS algorithm is making a trade between power consumption and user's requirement. under the prerequisite of satisfying the user's requirement, modified CFS will give some compensation to CPU bounded process. So that the new algorithm can save some energy in some case.

1.3 Research's architecture

We start with a discussion of the design of multicore processors in Chapter 2.1-2.2. This includes an overview of the concept of multicore processors. We then discuss the events leading to the development of multicore processors. We also present an overview of the architecture of multicore processors. Finally, we explain the concepts of homogeneous versus heterogeneous multicore processors. Then we discuss challenges associated with the implementation of multicore processors. We include a discussion of cache coherence, power and temperature management, and performance issues arising from the implementation of multicore processors.

In Chapter 3.1-3.3, this paper will introduce some common algorithm in Linux kernel in detail and then the analysis of each algorithm will be present, especially the disadvantages and shortcoming. Then, this research will raise the modified CFS and also the difficulties of implementing. In chapter 2.4-2.5, this paper will show the algorithm of modified CFS in detail and the C code of implementing, including the new system call of accessing the process control blocks, the usage of PAPI and complier of the new Linux kernel.

A discussion of the hardware implement is included in Chapter 4. Finally, we conclude and present some ideas for future research in Chapter 5.

2 MULTICORE PROCESSORS

2.1. Current Situation

2.1.1Overview

A multicore processor is comparable to having two or more separate processors installed on the same computer. However, because these processors are plugged into the same socket, the connection between them is faster. These cores are typically integrated onto a single integrated circuit die, known as a chip multiprocessor (CMP), or onto multiple dies in a single chip package

Refering to the integration of multiple processor cores on a processor chip, which each core is a separate microprocessor, the processor core can execute multiple threads in parallel. The multi-core processor may be a plurality of core may be heterogeneous with the structure, in this article we will focus on Heterogeneous multicore processors. The multi-core system is the use of multi-core processor computer systems, we expect the use of multi-core systems to maximize the use of thread-level and should conduct provide powerful computing power application

2.1.2 Development

"Moore's Law" specified in the field of semiconductors, along with the size of transistors getting smaller and smaller, the speed is faster and faster, the number of transistors we can integrate on a single chip is also increasing. Computer processor is the use of these additional resources to continuously improve transistor performance. But increasingly large and complex order processing structure led chip resource utilization continue to lower power consumption and increasing processor chip design and increasingly difficult to increase clock frequency.

The speedup of a program using a multicore processor in parallel computing is quantified using Amdahl's law. Amdahl's law states that the overall speedup of a program is

$$S = \frac{1}{(1-P) + \frac{P}{n}}$$

where P is the portion of the program that can be execute in parallel and n is the number of concurrent processors.

These difficulties mainly in the following two:

First, more and more difficult to improve the processor clock frequency. A large part of the microprocessor performance improvements is to rely on the progress of the manufacturing process, so that

components continue to shrink, while also decreasing the delay device, but the interconnect delay cannot sync between the device is reduced, so that more and further increase the clock frequency difficult. Therefore, in 2005 after the release of Intel's processor 3.8G Hz, we had to give up research and development And has launched 4G MHz processor clock frequency.

Second, the processor power is growing. With the improvement of the manufacturing process, the transistor size smaller, in on a chip integrated growing number of transistors, plus increase the clock frequency of the processor chip's power consumption is growing. But the chip space is limited, easily lead to overheating of the chip power consumption becomes large, reduce component stability.

Therefore, since the last century, researchers to look more and more to invest in multi-core processors. Many core processor originally developed by Computer Systems Laboratory at Stanford University proposed in 1996, which integrates multiple processor cores on a processor chip, each processor core parallel execution different threads, more similar to traditional symmetric multi-processor parallel system. Each of the multi-core processors processing core are the equivalent of a separate microprocessor, but the structure is more simple, easy-to-design and expansion, and less power. It is through the parallel execution of multiple threads, rather than increasing processor clock frequency to increase computing capacity, improve performance, and effective use of resources in the chip.

Multicore processor was originally concern and research in academia, including Stanford Hydra CMP project (Olukotun et al, 1996; Hammond et al, 2000), the University of Wisconsin Multiscalar project (Sohi et al, 1995; Gopal et al, 1998), Carnegie - Mellon University Stampede items (Steffan et al, 1998) and the Massachusetts Institute of M-machine project (Keckler et al, 1998). The first commercial multicore processors are released by IBM in 2001 Power4 Processors, each Power4 processor integrates two 64-bit PowerPC processing core 1G Hz, Followed by Hewlett-Packard introduced a similar multi-core PA-RISC 8800 processors in 2003, Sun has also pushed a dual-core UltraSPARC processor architecture, which is mainly used on high-end commercial server (Liu Will comfort et al., 2007).

With AMD and Intel in 2005 launched their commercial and dual-core processors Opteron Montecito, multi-core processors in our daily work and life rapidly gaining in popularity. Currently on the market The overwhelming majority of computer central processor multi-core processors, and integrated processing cores on a single chip An increasing number, the development trend of CPU chip structure is produced by monocytes (signal-core) to the multi-core (Multi-core) and then to many-core (many-core), a new piece of Moore's Law to predict a 18-24 per nucleus will be doubled

2.1.3 Architecture

A typical multi-core system architecture shown in the following figure. Core 1 to Core N to N treatments cores, each processing core has a private primary cache (L1 Cache), all processing cores share a last-level cache (Shared Last Level Cache), cache and main memory connected via a bus.

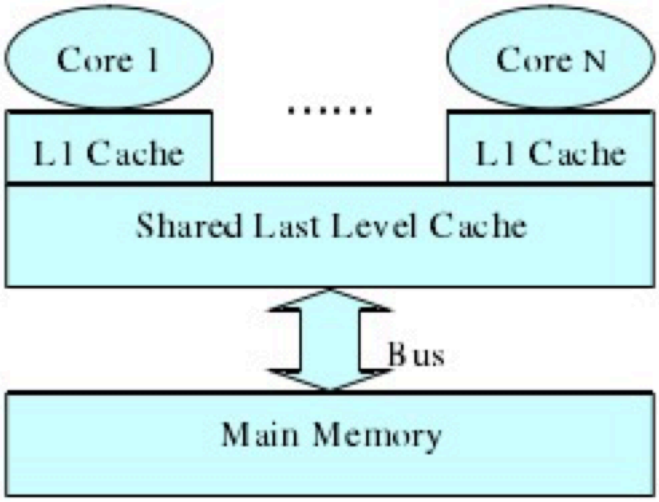

Figure 2.1

2.1.4 Homogeneous versus Heterogeneous Multicore Processors

Homogeneous multicore systems are systems with cores which have the same physical characteristic. A heterogeneous multicore processor instead uses a combination of different cores, each of which can be optimized for a different role. For example, core's operating frequency is the most obvious physical characteristic of cores.

A simply heterogeneous multicore processor, can consist of cores which run the same instruction set architecture, but with different performance. So that cores can give out power consumption but with the same tasks and programs. Asymmetry (or heterogeneity) can not only be built in designing, but also due to clock frequency scaling.

2.2 Challenges

There are numerous challenges of a multicore processor system. Three of these, which will be discussed in this chapter, are cache coherence, power and heat dissipation, and performance

2.2.1 Cache Coherence

Because of distributed level 1 and level 2 caches, the problem of Cache coherence has been enlarged. Every caches have their own copy of the data, so when copy of data has changed, the other copy of data in different caches cannot be updated right now, which cause that the data is out of time. If some other processes want to access the data, it has to wait until the data is updated, which means caches coherence. So if the cachees are not cohenrence, the overhead of data update will destory the performance of the sysytem.

2.2.2 Power and Temperature Management

Power consumption and heat dissipation must be taken consideration into the design and implementation of multi-core processor. In theory, if two cores are put at the same chip and without any design of reducing the heat disipation, the power consumption and heat dissipation will be doubled and even worse, because the heat will cause an extra heat dissipation. Besides, thermal constraints would make the problem worse. So the proper architecture must be concerned in order to reduce the extra heat and prolong the longevity of hardware.

2.2.3 Multi-core Performance

The performance of system will increase without any doubt. But if a task has very bad support of parallel computation, the gaining of performance will be reduced a lot and even has a worse performance than before. So the problem will be considered by every future software designer.

3 Algorithm for specific core

3.1. Common scheduler algorithm

3.1.1 overview

Process scheduling is the most critical function in a operating system. Scheduler is only a part of the procedure of scheduling, which is exactly complex and need many system work together. But this chapter focus on the scheduler and its algorithm. The main job of scheduler is selecting a proper process in the ready queue to be executed. As a general operating system, Linux scheduler can be divided into three categories:

1.interactive process

this kind of process has a large amount of interaction between user and computer, so the process is frequently at the state of sleep and wait for user's input. The typical application is editor vi. This kind of process has a high requirement of system's response time, otherwise user will feel system sluggish.

2. Batch process

This kind of process does not need interaction and always runs background, so that it need a large part ion resource. Thus, it can bear response delay, e.g. complier.

3. Real-time process

It has the highest requirement of response delay and always need tone executed at first. For example, video player. According to different categories, Linux has different strategy of scheduling. For real-time process, Linux use FIFO or round robin. For normal process, Linux will divide them into interaction or batch. Traditional Linux schedulers choose to improve the priority to let the process be select first. However, CFS focus on "completely fair". This idea not only simplifies the code, but also has a better performance in different scenario.

3.1.2 develop history

3.1.2.1 Scheduler in Linux2.4

Scheduler in Linux2.4.18 bases on priority and the algorithm of pick next is pretty easy: comparing the priority of all the processes in the ready queue one by one and pick the process with the highest priority as the next process to be executed. Besides, every process would be assign a time slice when it was forked. Clock interrupt decreases the time slice of the executing process, after the time slice is used up, the process has to wait until time slice is assigned again. And only after all the processes' time slices are used up, the time slices would be assign again. Every process has different requirement of scheduling, Linux2.4 scheduler satisfies the requirement according to the priority.

Real-time process's priority is set statically and always higher than normal process's, which means only when there is no real-time process in the queue, the normal process can be picked. For real-time process, there are two strategies: SCHED_FIFO and SCHED_RR. FIFO is first in first out, RR is round robin.
For normal process, interactive process has relatively higher priority. The priority of normal process is decided by the Counter field in process control block (also plus the nice value). When the child process is forked, its counter value would be half of its parent's, so that it can make sure that any process cannot use forking new process to get more chance of execution.

3.1.2.2 O(1) scheduler in Linux2.6

The time cost of O(1) scheduling algorithm is constant, not depend on the number of process. Besides, Linux2.6 supports kernel preemption, thus real-time process has better performance.
Linux2.6 kernel also has three scheduling strategy. SCHED_FIFO and SCHED_RR are used for real-time process, but SCHED_NORMAL is used for normal process. Also there are two modification of O(1) when it is compared to Linux2.4, one is the formula of priority calculation, the other is the algorithm of pick next.

3.1.2.2.1 Calculation of priority

Normal process's priority is calculated dynamically in the following formula:

dynamic priority = max (100, min (static priority – bonus +5, 139))

The bonus depends on its average of sleep time. So in Linux2.6, the longer sleep time, the larger bonus and higher priority.
Then if

Dynamic priority ≤ 3 x static priority/4 + 28

the process would be regarded as an interactive process. Average sleep time increases when it goes into sleep state and decreases when it goes into running state. The timing of update the value is: scheduler_tick(); fork(); awake from TASK_INTERRUPTIBLE and so on.
Real-time process's priority is set by sys_sched_setschedule(). The priority would not modify dynamically and always higher than normal process.

3.1.2.2.2 Algorithm of pick next

Normal process's algorithm of pick next is based on process's priority, higher priority means more chance to be picked.
Scheduler maintain two process queue array for one CPU: active array and expire array. In array, every element save one process queue point to one priority. Because of 140 different priorities, the length of arrays is 140.

When scheduler needs to pick the process with the highest priority, Linux2.6 schedulers does not need to iterate the whole ready queue but pick the first process in the queue with the highest priority from the active array. In order to achieve this algorithm, active array need to maintain a bitmap.

In order to improve the response time of interactive processes, O (1) scheduler not only improve processes' priority dynamically, but also adapt the following methods:
Every clock tick interrupt, process's time slice would minus 1. When time slice equals to 0, scheduler would judge the category of the process, if it is a interactive or real-time process, reset the time slice and insert it into active array again. If not, move the process from active array to expired array. So that interactive and real-time process always get CPU time first. Process cannot always stay at active array, otherwise the expire array would be starvation. So if the process has been executed over a threshold, even the process is an interactive or real-time process, the process would be moved to expired array.

When all the processes are moved to expire array, scheduler exchanges the active array and expire array. When process is moved to expire array, scheduler would reset its time slice, so the new active array would return to its initial state, and expire array would be empty, which means the new turn of scheduling begins.

3.1.3 new generation scheduler

3.1.3.1 staircase scheduler

There is a great different between algorithm of SD and O(1), SD abandon the conception of dynamic priority, but adapt an idea named as completely fair. The idea is pretty simple, but many tests show that the response of interactive process is much better former scheduler and extremely simplify the code. In the same token, SD also maintain an active array and pick the process. And the difference is that the process uses up its time slice, it would not be moved to expire array but moved to lower priority in active array, like a staircase. Once the process at the lowest staircase and uses up time slice, it would go back to its initial priority's queue.
The advantage of SD is that it deletes O (1) scheduler's complex code of dynamical priority; Eliminate expire array; The most important is that it prove completely fair is viable.

3.1.3.2 RSDL（The Rotating Staircase Deadline Schedule）

RSDL is an improvement of SD algorithm. The main idea is "completely fair". In addition, RSDL introduces expire array again. It assigns each priority a "grouping time" Tg; each process with the same priority has been assigned to the same time Tp(Tp is less than time slice). After Tp is used up, the process would be demoted to lower priority process group. The procedure is the same as SD, but it is called as minor rotation.
The following figure presents minor rotation.

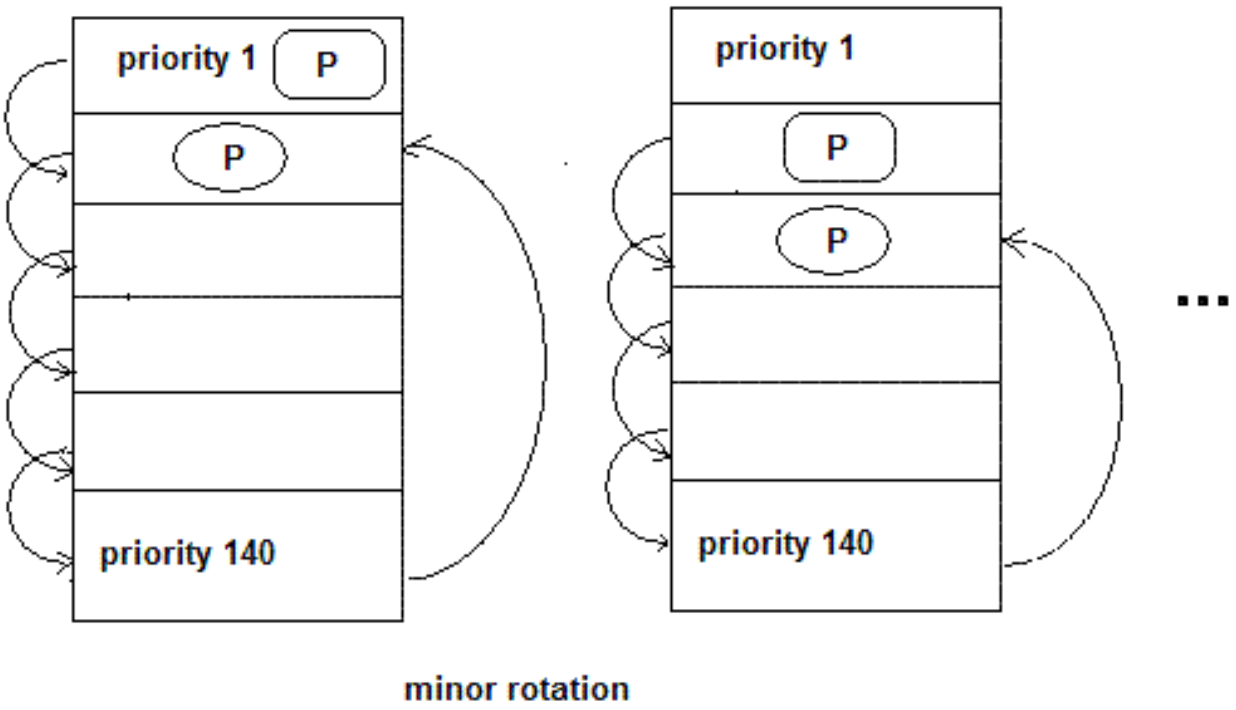

Figure 3.1

In SD algorithm, the process which is at the lowest staircase has to wait all the high priority processes until they finish. So the low priority process may wait for a long time. But in RSDL, high priority process group uses up Tg, no matter how the whole group would be demoted to lower priority process group. So that the waiting time becomes predictable. Process uses up its time slices (at T2 in the figure), it would be moved to expire array.

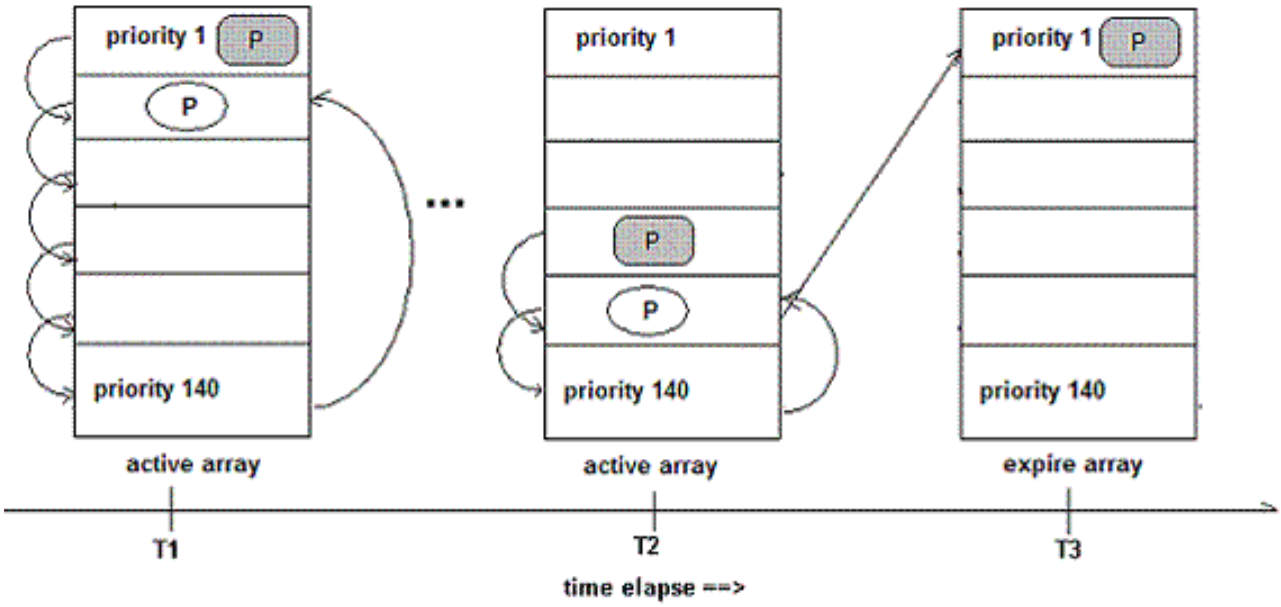

Figure 3.2

when active array is empty or all the process are demoted to lowest priority, major rotation is triggered. Major rotation is exchanging active array and expire array. All the processes return their initial states, and a new turn minor rotation begins.

3.1.4 CFS (completely fair scheduler)

3.1.4.1 overview

CFS will run each process for some amount of time, round-robin, selecting next the process that has run the least.

Rather than assign each process a time slice, CFS calculates how long a process should run as a function of the total number of runnable processes.

Instead of using the nice value to calculate a time slice, CFS uses the nice value to weight the proportion of processor a process is to receive: Higher valued (lower priority) processes receive a fractional weight relative to the default nice value, whereas lower valued (higher priority) processes receive a larger weight.

the number of runnable tasks approaches infinity, the proportion of allotted processor and the assigned time slice approaches zero. As this will eventually result in unacceptable switching costs, CFS imposes a floor on the time slice assigned to each process. This floor is called the minimum granularity. By defaults it is 1 millisecond

CFS uses vruntime to account for how long a process has run and thus how much longer it ought to run. When CFS is deciding what process to run next, it picks the process with the smallest vruntime.

For example

a scheduling period is called targeted latency (e.g. 20 ms). If there are two process with the same nice value, they will obtain 10 ms for each. So that every runnable process can be executed in one scheduling period.

3.1.4.2 The structure

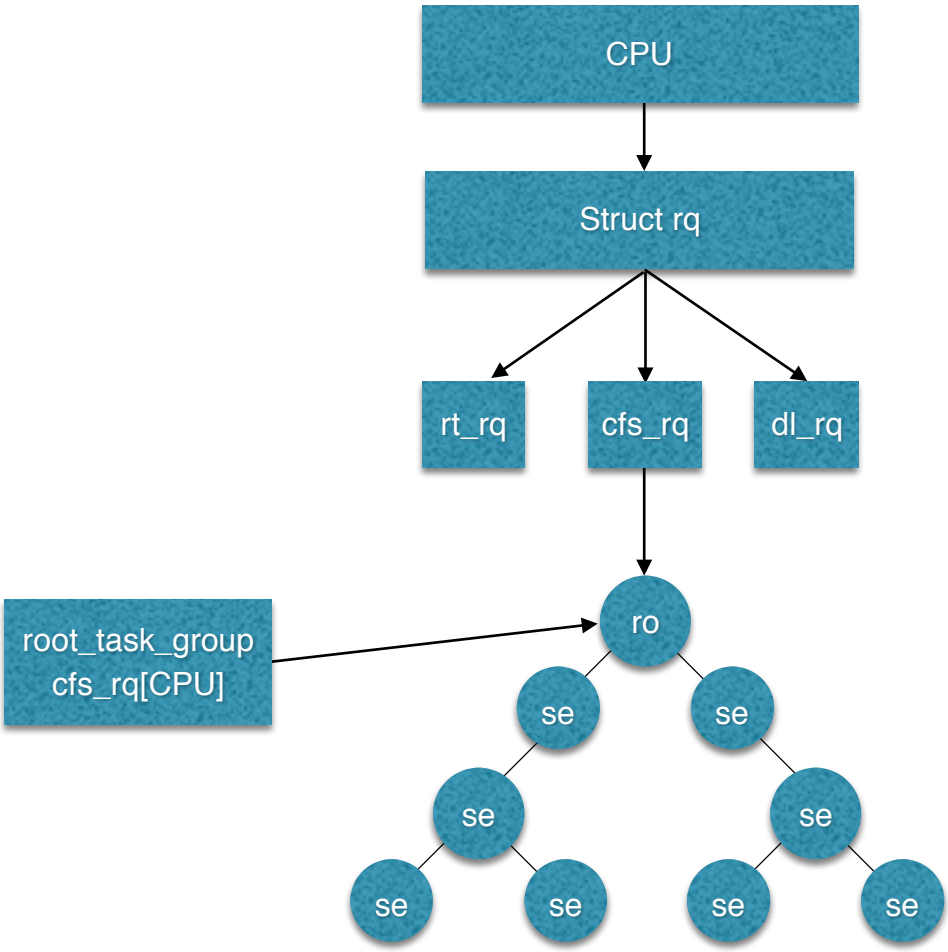

Figure 3.3

Every CPU has a runnable queue structure and every runnable queue structure has three runnable queue. One is real time runnable queue, one is cfs runnable queue and another is dl runnable queue. Every cfs runnable queue has a red&black tree.

3.1.4.3 process insert, pick & remove

3.1.4.3.1Picking

There is a red-black tree populated with every runnable process in the system where the key for each node is the runnable process's virtual runtime. Given this tree, the process that CFS wants to run next, which is the process with the smallest vruntime, is the leftmost node in the tree. The function that performs this selection is __pick_next_entity(), defined in kernel/sched_fair.c:

```
static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq) {
        struct rb_node *left = cfs_rq->rb_leftmost;
        if (!left)
                return NULL;
        return rb_entry(left, struct sched_entity, run_node);

}
```

__pick_next_entity() does not actually traverse the tree to find the leftmost node, because the value is cached by rb_leftmost. The return value from this function is the process that CFS next runs. If the function returns NULL, there is no leftmost node, and thus no nodes in the tree. In that case, there are no runnable processes, and CFS schedules the idle task.

3.1.4.3.2 Adding

when a process becomes runnable (wakes up) or is first created via fork(). Adding processes to the tree is performed by enqueue_entity():

```
static void enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
        /* Update the normalized vruntime before updating min_vruntime through callig update_curr().*/

        if (!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATE))
                se->vruntime += cfs_rq->min_vruntime;

        /*Update run-time statistics of the 'current'. */
        update_curr(cfs_rq);
        account_entity_enqueue(cfs_rq, se);

        if (flags & ENQUEUE_WAKEUP)
        {
                place_entity(cfs_rq, se, 0);
                enqueue_sleeper(cfs_rq, se);
```

```
        }

        update_stats_enqueue(cfs_rq, se);
        check_spread(cfs_rq, se);
        if (se != cfs_rq->curr)
                __enqueue_entity(cfs_rq, se);
}
```

This function updates the runtime and other statistics and then invokes __enqueue_entity() to perform the actual inserting the entry into the red-black tree:

```
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
        struct rb_node **link = &cfs_rq->tasks_timeline.rb_node;
        struct rb_node *parent = NULL;
        struct sched_entity *entry;
        s64 key = entity_key(cfs_rq, se);
        int leftmost = 1;

        /*Find the right place in the rbtree: */
        while (*link) {
                parent = *link;
                entry = rb_entry(parent, struct sched_entity, run_node);
                /* We dont care about collisions. Nodes with * the same key stay together.*/
                if (key < entity_key(cfs_rq, entry)) {
                        link = &parent->rb_left;
                } else {
                        link = &parent->rb_right;
                        leftmost = 0;
                }
         }
        /*Maintain a cache of leftmost tree entries (it is frequently * used):*/
        if (leftmost)
                cfs_rq->rb_leftmost = &se->run_node;
        rb_link_node(&se->run_node, parent, link);
        rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
```

}

3.1.4.3.3Remove

CFS removes processes from the red-black tree.This happens when a process blocks (becomes unrunnable) or terminates (ceases to exist):

```
static void dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int sleep)
{
        /*Update run-time statistics of the 'current'. */
        update_curr(cfs_rq);

        update_stats_dequeue(cfs_rq, se);
        clear_buddies(cfs_rq, se);

        if (se != cfs_rq->curr) __dequeue_entity(cfs_rq, se);
        account_entity_dequeue(cfs_rq, se);
        update_min_vruntime(cfs_rq);

        /* Normalize the entity after updating the min_vruntime because the update can refer to the ->curr item
and we need to reflect this movement in our normalized position. */

        if (!sleep)
                se->vruntime -= cfs_rq->min_vruntime;
}
```

the real work is performed by a helper function, __dequeue_entity():

```
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se) {
        if (cfs_rq->rb_leftmost == &se->run_node) {
                struct rb_node *next_node;
                next_node = rb_next(&se->run_node);
                cfs_rq->rb_leftmost = next_node;
        }
        rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
```

}

rb_erase() function performs all the work.The rest of this function updates the rb_leftmost cache. If the process-to-remove is the leftmost node, the function invokes rb_next() to find what would be the next node in an in-order traversal. This is what will be the leftmost node when the current leftmost node is removed.

3.1.4.4 Detail of CFS

The Virtual Runtime, declaration vruntime is a variable part of the process inherited C structure as defined in linux/sched.h which accounts for the time a process run in relation to the number of running processes. Each

```
struct task_struct {
            volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
            void *stack;
            atomic_t usage;
            unsigned int flags;     /* per process flags, defined below */
            unsigned int ptrace;
    (...)
    #endif
            int on_rq;
            int prio, static_prio, normal_prio;
            unsigned int rt_priority;
            const struct sched_class *sched_class;
            struct sched_entity se;
```

Figure 3.4

process holds a process descriptor “task_struct” dynamically allocated via the slab allocator.

```
struct sched_entity {
        struct load_weight      load;           /* for load-balancing */
        struct rb_node          run_node;
        struct list_head        group_node;
        unsigned int            on_rq;

        u64                     exec_start;
        u64                     sum_exec_runtime;
        u64                     vruntime;
        u64                     prev_sum_exec_runtime;

        u64                     nr_migrations;

#ifdef CONFIG_SCHEDSTATS
        struct sched_statistics statistics;
#endif

#ifdef CONFIG_FAIR_GROUP_SCHED
        struct sched_entity     *parent;
        /* rq on which this entity is (to be) queued: */
        struct cfs_rq           *cfs_rq;
        /* rq "owned" by this entity/group: */
        struct cfs_rq           *my_q;
#endif
};
```

Figure 3.5

The sched_entity structure linked in “task_struct” is defined as

CFQ does not account time slice as did previous schedulers, the vruntime is evaluated

The vruntime for each process is calculated as followed:

1. Compute the time spent by the process on the CPU
2. Weight the computed running time against the number of runnable processes

The update_curr function is responsible to calculate the running time spent by the process, which stores the value

```
 delta_exec = (unsigned long)(now - curr->exec_start);
with
 struct sched_entity *curr = cfs_rq->curr;
 u64 now = rq_of(cfs_rq)->clock_task;
```

Figure 3.6

into an unsigned long variable delta_exec which is computed as followed:

```
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
              unsigned long delta_exec)
{
        unsigned long delta_exec_weighted;

        schedstat_set(curr->statistics.exec_max,
                      max((u64)delta_exec, curr->statistics.exec_max));

        curr->sum_exec_runtime += delta_exec;
        schedstat_add(cfs_rq, exec_clock, delta_exec);
        delta_exec_weighted = calc_delta_fair(delta_exec, curr);

        curr->vruntime += delta_exec_weighted;
        update_min_vruntime(cfs_rq);

#if defined CONFIG_SMP && defined CONFIG_FAIR_GROUP_SCHED
        cfs_rq->load_unacc_exec_time += delta_exec;
#endif
}
```

Figure 3.7

delta_exec is passed unto __update_curr

calc_delta_fair will return the weighted value of the process's calculated runtime delta_exec in relation to number of runnable processes. The sub function used for that calculation is calc_delta_mine.

unsigned long delta_exec, is the computed running time of the process

unsigned long weight, is the nice value of the process

```
calc_delta_mine(unsigned long delta_exec, unsigned long weight,
                struct load_weight *lw)
{
        u64 tmp;

  /*
   * weight can be less than 2^SCHED_LOAD_RESOLUTION for task group sched
   * entities since MIN_SHARES = 2. Treat weight as 1 if less than
   * 2^SCHED_LOAD_RESOLUTION.
   */
        if (likely(weight > (1UL << SCHED_LOAD_RESOLUTION)))
                tmp = (u64)delta_exec * scale_load_down(weight);
        else
                tmp = (u64)delta_exec;

        if (!lw->inv_weight) {
                unsigned long w = scale_load_down(lw->weight);

                if (BITS_PER_LONG > 32 && unlikely(w >= WMULT_CONST))
                        lw->inv_weight = 1;
                else if (unlikely(!w))
                        lw->inv_weight = WMULT_CONST;
                else
                        lw->inv_weight = WMULT_CONST / w;
        }


        if (unlikely(tmp > WMULT_CONST))
               tmp = SRR(SRR(tmp, WMULT_SHIFT/2) * lw->inv_weight,
                        WMULT_SHIFT/2);
        else
                tmp = SRR(tmp * lw->inv_weight, WMULT_SHIFT);

        return (unsigned long)min(tmp, (u64)(unsigned long)LONG_MAX);
}
```

Figure 3.8

struct load_weight *lw, composed of 2 unsigned long “weight” and “inv_weigh” (lw->weight and lw->inv_weight)

```
curr->vruntime += delta_exec_weighted;
```

Figure 3.9

Once the vruntime is computed, it is stored into the inherited sched_entity structure of the process by calling

```
update_min_vruntime(cfs_rq);
```

Figure 3.10

in the previously seen __update_curr function, we also notice the call
What this does is to compute the smallest vruntime of all runnable processes and store it at the leftmode node of the red-black tree.

The CFS selection algorithm for process to be run is extremely simple, it will run its red black tree and search for the smallest vruntime value pointing to the next runnable process. In other words, “run the process which is represented by the leftmost node in the tree”, since the leftmost node constains the smallest vruntime, referred in the source code as min_vruntime.

To conclude this post, it is important to note that CFS does not walk the whole tree since min_vruntime is

```
struct cfs_rq {
        struct load_weight load;
       unsigned long nr_running, h_nr_running;

        u64 exec_clock;
        u64 min_vruntime;
#ifndef CONFIG_64BIT
        u64 min_vruntime_copy;
#endif

        struct rb_root tasks_timeline;
        struct rb_node *rb_leftmost
```

Figure 3.11

referenced by rb_leftmost in the cfs_rq struct (kernel/sched.c)

```c
static struct sched_entity *__pick_next_entity(struct sched_entity *se)
{
        struct rb_node *next = rb_next(&se->run_node);

        if (!next)
                return NULL;

        return rb_entry(next, struct sched_entity, run_node);
}
```

Figure 3.12

as seen in this construct (kernel/sched_fair.c)

3.2. Shortcoming

3.2.1 Linux 2.4

Bad Expansibility：Scheduler need to iterate the whole ready queue to pick the proper process, so the cost of time of the algorithm depends on the number of processes. Also maintenance of counter has a large overhead especially when the number of processes in system is high, which would cause extremely decrease of performance. Performance of scheduling is low in high load system: the scheduler pre-assign each process a relatively large time slice, thus in a high load server, the efficiency of scheduler is low.

Optimization of interactive is not good: Scheduler always assumes that interactive process always as SUSPENDED state. In fact, some batch process also frequently at SUSPENDED state for IO operation. For example, a database engine is querying, it is frequently doing disk IO. Even if it does not require high user response, it is also increased the priority.

Support of real-time process is not enough: Linux2.4 kernel is not preemption, which is not acceptable for real-time process.

3.2.2 O(1) scheduler

High Complexity of code: For example, the following programs always make bad performance of scheduler and cause slow response of interactive process: fiftyp.c, thud.c, chew.c, ring-test.c, massive_intr.c. Difficult maintenance of code: Calculation of dynamic priority, average sleeping time and some obscure formula to revise priority and differentiate interactive process.

3.2.3 Completely fair

SD, RSDL and CFS all adapt the idea of completely fair and many tests shows that the idea and CFS algorithm have very good performance. But CFS does not take power consumption into consideration. For example, there are two processes, one is a CPU bounded process A with low priority and the other is IO bounded process B with higher priority. So process B has more chance to be picked at first and larger time slice, which cause that B cannot take a good use of CPU time, because B has to wait for IO. In other words, CPU has more idle time. Although during idle time the power consumption is much less than busy time, for laptop and PDA, the consumption cannot be ignored.

So this chapter would talk about a modified CFS algorithm to make it distinguish CPU bounded and IO bounded process and give some punish to IO bounded process, so that the new algorithm can reduce some waiting time.

3.3. Improvement and difficulties

3.3.1 Improvement

3.3.1.1 Principle

For CFS:
A is IO bounded process with high priority
B is CPU bounded process with low priority
|<————-A————->|<——-B——>|<————-A-————>|<—-B-—>|

run	wait	run	run	w	run	wait	run	run

|--5ms--|—5ms--|—5ms-|—10ms—---|2ms|-5ms---|-5ms—|—5ms--|—10ms—---|
So the total waiting time is 5+2+5=12ms
total running time is 15+12+15+10=52ms
ratio=0.23

For CFS with some punishes to IO bounded process
A is IO bounded process with high priority
B is CPU bounded process with low priority
Scheduler give some punishes to A but does not change the actual priority
|<————-A————->|<————————B———>|<————-A-————>|<—B-—>|<—A->

run	wait	run	run	w	run	run	wait	run	run

|--5ms--|—5ms--|—2ms-|——-10ms——-----|2ms|-3ms-|——8ms——---|—4ms—|—7ms-——---|-5ms—|
So the total waiting time is 5+2+4=11ms
total running time is 12+15+12+7+5=51ms

ratio=0.21

So obviously, if the scheduler can punish the IO processes, not only the waiting time is reduce, but also the total running time will also reduce. In other words, the new CFS can save some power consumption in some case.

3.3.1.2 IO, CPU bounded judement

A program is CPU bound if it would go faster if the CPU were faster, i.e. it spends the majority of its time simply using the CPU (doing calculations). A program that computes new digits of π will typically be CPU-bound, it's just crunching numbers.

A program is I/O bound if it would go faster if the I/O subsystem was faster. Which exact I/O system is meant can vary; I typically associate it with disk. A program that looks through a huge file for some data will often be I/O bound, since the bottleneck is then the reading of the data from disk. In other words, we can find that IO bound process has large amount of context switch and high cache miss rate and even high number of CPU migration.

So in this modified CFS algorithm, we would introduce cache miss rate to punish IO bound process and compensate CPU bound process, in this way the chance of first picking of IO bound process with high priority would lower and the time slice will be decreased. In the same token, CPU bound process will profit. Finally, the whole system will also profit from the new mechanism.

3.3.1.3 Idea of implement

3.3.1.3.1 overview

First of all, I would insert a new attribute into process control block named as cache miss rate and a new attribute in struct cfs_rq named as average cache miss rate. Then I would initialize cache miss rate in /kernel/sched/core.c and initialize average cache miss rate in /kernel/sched/fair.c. Then the process will calculate its cache miss rate and feed it both into cache miss rate and average miss rate. Finally, use cache miss rate and average cache miss rate to calculate the compensation and the punishment.

3.3.1.3.2 formula

a. use the new feed data to update the cache miss rate

now we define:

cache_miss_rate_past;

number_cache_miss_past;

average_cache_miss_queue_past;

number_cache_miss_queue_past;

execute_time_past;

number_process_queue;

number_cache_miss_new_turn;

new_turn_time;

the first six variables are maintained in the process control block and the last two variables are the new feed data.

So the new cache miss rate is calculated by:

(number_cache_miss_past+number_cache_miss_new_turn)/(new_turn_time+execute_time_past);

And the new average cache miss rate of a ready queue is:

(average_cache_miss_queue_past*number_process_queue*execute_time_past+number_cache_miss_new_turn)/((new_turn_time+execute_time_past)*number_process_queue)

It is obvious that the formula make the system has memory and use the past value to evaluate the new value. Maybe it looks fair and overview, but it makes our system need more overhead to calculate the value and more space to save the data.

So I decide to make the system memoryless, the formula becomes:

we only define:

cache_miss_rate_past;

average_cache_miss_queue_past;

number_process_queue;

number_cache_miss_new_turn;

new_turn_time;

the first three variables are maintained in the process control block and the last two variable is the new feed data.

So the new cache miss rate is calculated by:

(number_cache_miss_new_turn)/(new_turn_time);

And the new average cache miss rate of a ready queue is:

(average_cache_miss_queue_past*number_process_queue+number_cache_miss_new_turn)/(new_turn_time))/(number_process_queue)

b. compensate value

we define:

cache_miss_rate;

average_cache_miss_queue;

load;

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */ 9548, 7620, 6100, 4904, 3906,
    /* -5 */ 3121, 2501, 1991, 1586, 1277,
    /* 0 */ 1024, 820, 655, 526, 423,
    /* 5 */ 335, 272, 215, 172, 137,
    /* 10 */ 110, 87, 70, 56, 45,
    /* 15 */ 36, 29, 23, 18, 15,
};
```

```
/*
* Nice levels are multiplicative, with a gentle 10% change for every
* nice level changed. I.e. when a CPU-bound task goes from nice 0 to
* nice 1, it will get ~10% less CPU time than another CPU-bound task
* that remained on nice 0.
*
* The "10% effect" is relative and cumulative: from _any_ nice level,
* if you go up 1 level, it's -10% CPU usage, if you go down 1 level
* it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
* If a task goes up by ~10% and another task goes down by ~10% then
the relative distance between them is ~25%.)
```

Figure 3.13

3.3.2 Difficulties

3.3.2.1 change load

Even if there is map to link the nice value and load, but in the kernel scheduler will use the load to calculate vruntime and time slice very frequently. So system change the former formula into

$$\text{Delta vruntime}=\text{deltaTime}*1024*\frac{2^{32}}{load*2^{32}}=(\text{deltaTime}*1024*\text{inv\_load})>>32$$

so we cannot change the load directly and we can only change the delta vruntime and time slice.

3.3.2.2 get the cache miss rate

If we want to get the cache miss rate of one process, we need to access to the register and get the data from register. There exists some command to retrieve the data from command line, like perf and so on. But if we want to call the function in the kernel to get data is extremely different.

And there is another way to get the data: we can let the process calculate its own cache miss rate at the user kernel and call a system call to feed the data into process control block. But the question is how can we feed the data from user level to kernel level.

3.3.2.3 float type data

As we know in the linux kernel, the float type data is disabled. So we can enable SSE to make kernel do calculate between different kind of data type. Or we can use IEEE 754 to transfer float number to hex-number and use special operator to do calculation. Or we need to imitate the float data in the kernel.

3.3.3 Solutions

3.3.3.1 Overview

Finally, we choose to get the cache miss rate form user level and feed it to kernel.

a. we install the PAPI library to let PAPI monitor the hardware performance. It can measure the number of L1/L2 cache miss/access in a given time.

b. we will write two new system call to the system call table. One help us access to process control block and the other will retrieve the data which we feed so that can help us make sure whether our system calls work well.

c. Instead of changing the load directly, we choose to modify the vruntime and time slice respectively with the same coefficient.

3.3.3.2 PAPI

PAPI(Performance Application Programming Interface) is a multi-platform library for portably accessing hardware counters for event profiling of software including flop counts, cache efficiency, and branch prediction rates.

PAPI provides two interfaces to the underlying counter hardware; a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level interface directed towards users with more sophisticated needs. The low level PAPI interface deals with hardware events in groups called *EventSets*. EventSets reflect how the counters are most frequently used, such as taking simultaneous measurements of different hardware events and relating them to one another. For example, relating cycles to memory references or flops to level 1 cache misses can indicate poor locality and memory management. In addition, EventSets allow a highly efficient implementation which translates to more detailed and accurate measurements. EventSets are fully programmable and have features such as guaranteed thread safety, writing of counter values, multiplexing and notification on threshold crossing, as well as processor specific features. The high level interface simply provides the ability to start, stop and read specific events, one at a time.

We can call the following function in the PAPI library to get the hardware performance:
PAPI_library_init
PAPI_create_eventset

PAPI_add_event / PAPI_add_events

PAPI_remove_event PAPI_remove_events

PAPI_start

PAPI_read

PAPI_stop

PAPI_cleanup_eventset

PAPI_destroy_eventset

PAPI_shutdown

The concrete code will be shown in the next subchapter.

3.3.3.3 Imitate float type data

co=cache_miss_rate/average_cache_miss_queue;

But kernel cannot handle the float data, so co is always a long type data, which makes the calculation is nor precise. Thus we define:

co_1 // the integer part of co;

co_2 // decile

co_3 // prcentile

and

co_1=cache_miss_rate/average_cache_miss_queue;

co_2=10*cache_miss_rate/average_cache_miss_queue-10*co_1;

co_3=100*cache_miss_rate/average_cache_miss_queue-100*co_1-10*co_2;

when we use co to calculate the new time slice and vruntime:

delta_vruntime=runtime*co_1+runtime*co_2/10+runtime*co_3/100;

3.4. Algorithm Implement and Code

3.4.1 create new variable in process control block and struct cfs_rq

/include/linux/sched.h

sched_entity is a inner struct of task_struct(process control block)

```
1250  struct sched_entity {
1251      struct load_weight  load;       /* for load-balancing */
1252      struct rb_node      run_node;
1253      struct list_head    group_node;
1254      unsigned int        on_rq;
1255
1256      u64         exec_start;
1257      u64         sum_exec_runtime;
1258      u64         vruntime;
1259      u64         prev_sum_exec_runtime;
1260
1261      u64         nr_migrations;
1262
1263  #ifdef CONFIG_SCHEDSTATS
1264      struct sched_statistics statistics;
1265  #endif
1266
1267  #ifdef CONFIG_FAIR_GROUP_SCHED
1268      int         depth;
1269      struct sched_entity *parent;
1270      /* rq on which this entity is (to be) queued: */
1271      struct cfs_rq       *cfs_rq;
1272      /* rq "owned" by this entity/group: */
1273      struct cfs_rq       *my_q;
1274  #endif
1275
1276  #ifdef CONFIG_SMP
1277      /*
1278       * Per entity load average tracking.
1279       *
1280       * Put into separate cache line so it does not
1281       * collide with read-mostly values above.
1282       */
1283      struct sched_avg    avg ____cacheline_aligned_in_smp;
1284  #endif
1285      long cache_miss_rate;
1286      long co_1;
1287      long co_2;
1288      long co_3;
1289  };
```

Figure 3.14

kernel/sched/sched.h

```
360  struct cfs_rq {
361      struct load_weight load;
362      unsigned int nr_running, h_nr_running;
363
364      u64 exec_clock;
365      u64 min_vruntime;
366  #ifndef CONFIG_64BIT
367      u64 min_vruntime_copy;
368  #endif
369
370      struct rb_root tasks_timeline;
371      struct rb_node *rb_leftmost;
372
373      /*
374       * 'curr' points to currently running entity on this cfs_rq.
375       * It is set to NULL otherwise (i.e when none are currently running).
376       */
377      struct sched_entity *curr, *next, *last, *skip;
378
379  #ifdef  CONFIG_SCHED_DEBUG
380      unsigned int nr_spread_over;
381  #endif
382
383  #ifdef CONFIG_SMP
384      /*
385       * CFS load tracking
386       */
387      struct sched_avg avg;
388      u64 runnable_load_sum;
389      unsigned long runnable_load_avg;
390      //test
391      long runnable_cachemiss_avg;
392  #ifdef CONFIG_FAIR_GROUP_SCHED
393      unsigned long tg_load_avg_contrib;
394  #endif
395      atomic_long_t removed_load_avg, removed_util_avg;
396  #ifndef CONFIG_64BIT
397      u64 load_last_update_time_copy;
398  #endif
399
```

Figure 3.15

3.4.2 initialize the new variable

kernel/sched/core.c, initialize the sched_entity

```
2197  static void __sched_fork(unsigned long clone_flags, struct task_struct *p)
2198  {
2199      p->on_rq            = 0;
2200
2201      p->se.on_rq         = 0;
2202      p->se.exec_start        = 0;
2203      p->se.sum_exec_runtime      = 0;
2204      p->se.prev_sum_exec_runtime = 0;
2205      p->se.nr_migrations     = 0;
2206      p->se.vruntime          = 0;
2207      p->se.cache_miss_rate       = 1;
2208      p->se.co_1      = 1;
2209      p->se.co_2      = 0;
2210      p->se.co_3      = 0;
2211      INIT_LIST_HEAD(&p->se.group_node);
2212
2213  #ifdef CONFIG_FAIR_GROUP_SCHED
2214      p->se.cfs_rq            = NULL;
2215  #endif
2216
2217  #ifdef CONFIG_SCHEDSTATS
2218      memset(&p->se.statistics, 0, sizeof(p->se.statistics));
2219  #endif
2220
2221      RB_CLEAR_NODE(&p->dl.rb_node);
2222      init_dl_task_timer(&p->dl);
2223      __dl_clear_params(p);
2224
2225      INIT_LIST_HEAD(&p->rt.run_list);
2226
2227  #ifdef CONFIG_PREEMPT_NOTIFIERS
2228      INIT_HLIST_HEAD(&p->preempt_notifiers);
2229  #endif
```

Figure 3.16

kernel/sched/fair.c, initialize the cfs_rq;

```
8219  void init_cfs_rq(struct cfs_rq *cfs_rq)
8220  {
8221      cfs_rq->tasks_timeline = RB_ROOT;
8222      cfs_rq->min_vruntime = (u64)(-(1LL << 20));
8223
8224  #ifndef CONFIG_64BIT
8225      cfs_rq->min_vruntime_copy = cfs_rq->min_vruntime;
8226  #endif
8227  #ifdef CONFIG_SMP
8228      atomic_long_set(&cfs_rq->removed_load_avg, 0);
8229      atomic_long_set(&cfs_rq->removed_util_avg, 0);
8230  #endif
8231      cfs_rq->runnable_cachemiss_avg = 1;
8232  }
```

Figure 3.17

3.4.3 new formula for compensation and punishment

kernel/sched/fair.c

```
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;
    long co_1;
    long co_2;
    long co_3;

    if (unlikely(!curr))
        return;

    delta_exec = now - curr->exec_start;
    if (unlikely((s64)delta_exec <= 0))
        return;

    curr->exec_start = now;

    schedstat_set(curr->statistics.exec_max,
              max(delta_exec, curr->statistics.exec_max));

    curr->sum_exec_runtime += delta_exec;
    schedstat_add(cfs_rq, exec_clock, delta_exec);

    co_1=(curr->cache_miss_rate)/(cfs_rq->runnable_cachemiss_avg);
    co_2=10*(curr->cache_miss_rate)/(cfs_rq->runnable_cachemiss_avg)-10*co_1;
    co_3=100*(curr->cache_miss_rate)/(cfs_rq->runnable_cachemiss_avg)-100*co_1-10*co_2;
    if(co_1>1){
        co_1=1;
        co_2=2;
        co_3=5;
    }
    else if(co_1=1){
        if(co_2>2){
            co_2=2;
            co_3=5;
        }
        else if(co_2=2){
            if(co_3>5)
                co_3=5;
        }
        else;
    }
    else;
    if(co_1=0){
        if(co_2<8)
            co_2=8;
        else if(co_2=8)
            co_3=0;
        else;
    }
    curr->vruntime += calc_delta_fair(delta_exec, curr)*co_1+
    calc_delta_fair(delta_exec, curr)*co_2/10+
    calc_delta_fair(delta_exec, curr)*co_3/100;

    update_min_vruntime(cfs_rq);

```

Figure 3.18

3.4.4 new system call

3.4.4.1 add new system calls in system call table

arch/X86/entry/syscalls/syscall_64.tbl

```
316 common  renameat2       sys_renameat2
317 common  seccomp         sys_seccomp
318 common  getrandom       sys_getrandom
319 common  memfd_create        sys_memfd_create
320 common  kexec_file_load     sys_kexec_file_load
321 common  bpf         sys_bpf
322 64  execveat        stub_execveat
323 common  userfaultfd     sys_userfaultfd
324 common  membarrier      sys_membarrier
325 common  mlock2          sys_mlock2
326 common  copy_file_range     sys_copy_file_range
327 common  access_pcb      sys_access_pcb
328 common  retrive_pcb     sys_retrive_pcb

#
# x32-specific system call numbers start at 512 to avoid cache impact
# for native 64-bit operation.
#
512 x32 rt_sigaction        compat_sys_rt_sigaction
513 x32 rt_sigreturn        stub_x32_rt_sigreturn
514 x32 ioctl           compat_sys_ioctl
515 x32 readv           compat_sys_readv
516 x32 writev          compat_sys_writev
517 x32 recvfrom        compat_sys_recvfrom
518 x32 sendmsg         compat_sys_sendmsg
519 x32 recvmsg         compat_sys_recvmsg
520 x32 execve          stub_x32_execve
521 x32 ptrace          compat_sys_ptrace
522 x32 rt_sigpending       compat_sys_rt_sigpending
```

Figure 3.19

3.4.4.2 define new system call

include/linux/syscall.h

```
asmlinkage long sys_kcmp(pid_t pid1, pid_t pid2, int type,
            unsigned long idx1, unsigned long idx2);
asmlinkage long sys_finit_module(int fd, const char __user *uargs, int flags);
asmlinkage long sys_seccomp(unsigned int op, unsigned int flags,
                const char __user *uargs);
asmlinkage long sys_getrandom(char __user *buf, size_t count,
                  unsigned int flags);
asmlinkage long sys_bpf(int cmd, union bpf_attr *attr, unsigned int size);

asmlinkage long sys_execveat(int dfd, const char __user *filename,
            const char __user *const __user *argv,
            const char __user *const __user *envp, int flags);

asmlinkage long sys_membarrier(int cmd, int flags);
asmlinkage long sys_copy_file_range(int fd_in, loff_t __user *off_in,
                    int fd_out, loff_t __user *off_out,
                    size_t len, unsigned int flags);

asmlinkage long sys_mlock2(unsigned long start, size_t len, int flags);

#endif

asmlinkage long sys_access_pcb(pid_t pid,unsigned long nr_cachemiss, unsigned long delta_time);
asmlinkage long sys_retrive_pcb(pid_t pid,int type);
```

Figure 3.20

3.4.4.3 write function body of new system call

kernel/access_pcb.c

```c
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/syscalls.h>
#include "sched/sched.h"

asmlinkage long sys_access_pcb(pid_t pid,unsigned long nr_cachemiss, unsigned long delta_time){
    /*'in the past' means 'from very beginning to last turn of calculating the cache rate miss'
      'now' means ''in the past'+this turn of calculating the cache miss rate'

       for example, every 10ms, papi calculates the number of cache miss and call this system call to feed data.
           suppose that now is 100ms and data in the pcb has noty been updated, the procedure would be:
        1. retrive the data in the past(at 90ms)
            2. get the data calculated by papi in the new 10ms(from 90ms to 100ms)
            3. use 1. and 2. to calculating exactly data at 100ms
        4. update data in pcb
    */
    struct task_struct *tsk;
    struct sched_entity *se;
    struct cfs_rq* cfs_rq;

    unsigned long sum_p;//total cache miss of a process in the past
    long avg_p;//cache miss rate of a process in the past
    unsigned long nr;//total cache miss of a process from it was forked to now
    long rate;//cache miss rate of a process at now

    unsigned long sum_q;//total cache miss of all processes in queue in the past
    long avg_q;//cache miss rate of a queue in the past
    unsigned long nr_q;//total cache miss of all processes in queue from very beginning to now
    long rate_q;//cache miss rate of a queue at now

    rcu_read_lock();
    tsk = find_task_by_vpid(pid);
    if (tsk){
        get_task_struct(tsk);
        se=&(tsk->se);
        if(se)
            cfs_rq=(se->cfs_rq);
    }
    rcu_read_unlock();


        sum_p=se->cache_miss_nr;
        avg_p=se->cache_miss_rate;
        nr=sum_p+nr_cachemiss;
        rate=nr/(delta_time+(sum_p/avg_p));
        se->cache_miss_rate=rate;
        se->cache_miss_nr=nr;

    if(cfs_rq){
        sum_q=cfs_rq->runnable_cachemiss_sum;
        avg_q=cfs_rq->runnable_cachemiss_avg;
        nr_q=sum_q+nr_cachemiss;
        rate_q=nr_q/((sum_q/(avg_q*(cfs_rq->nr_running)))+delta_time);
        cfs_rq->runnable_cachemiss_sum=nr_q;
        cfs_rq->runnable_cachemiss_avg=rate_q;
    }


    put_task_struct(tsk);

    return 0;
}

```

Figure 3.21

kernel/retrive_pcb.c

```c
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/syscalls.h>
#include "sched/sched.h"

asmlinkage long sys_retrive_pcb(pid_t pid,int type){

    long retrive;
    struct task_struct *tsk;
    struct sched_entity *se;
    struct cfs_rq* cfs_rq;

    rcu_read_lock();
    tsk = find_task_by_vpid(pid);
    if (tsk){
        get_task_struct(tsk);
        se=&(tsk->se);
        if(se)
            cfs_rq=(se->cfs_rq);
    }
    rcu_read_unlock();

    if(type==1)
        retrive=se->cache_miss_rate;
    else if(type==2)
        retrive=se->cache_miss_nr;
    else if(type==3)
        retrive=cfs_rq->runnable_cachemiss_sum;
    else if(type==4)
        retrive=cfs_rq->runnable_cachemiss_avg;
    else
        retrive=cfs_rq->nr_running;
    put_task_struct(tsk);

    return retrive;

}

```

Figure 3.22

3.4.4.4 makefile

```
#
# Makefile for the linux kernel.
#

obj-y     = fork.o exec_domain.o panic.o \
            cpu.o exit.o softirq.o resource.o \
            sysctl.o sysctl_binary.o capability.o ptrace.o user.o \
            signal.o sys.o kmod.o workqueue.o pid.o task_work.o \
            extable.o params.o \
            kthread.o sys_ni.o nsproxy.o \
            notifier.o ksysfs.o cred.o reboot.o \
            async.o range.o smpboot.o sys_access_pcb.o sys_retrive_pcb.o

obj-$(CONFIG_MULTIUSER) += groups.o

ifdef CONFIG_FUNCTION_TRACER
# Do not trace debug files and internal ftrace files
CFLAGS_REMOVE_cgroup-debug.o = $(CC_FLAGS_FTRACE)
CFLAGS_REMOVE_irq_work.o = $(CC_FLAGS_FTRACE)
endif
```

Figure 3.23

3.4.4.5 PAPI test

```
/* Initialize the PAPI library */
if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
        exit(1);

/*Initialize the papi thread support*/
if (PAPI_thread_init(pthread_self) != PAPI_OK)
        exit(1);

/*Add another event*/
retval = PAPI_add_event(EventSet, PAPI_TOT_INS);
assert(retval==PAPI_OK);

/* Start counting events */
if (PAPI_start(EventSet) != PAPI_OK)
        retval = PAPI_start (EventSet);
assert(retval==PAPI_OK);

/* Clean up EventSet */
if (PAPI_cleanup_eventset(EventSet) != PAPI_OK)
        exit(-1);

/* Destroy the EventSet */
if (PAPI_destroy_eventset(&EventSet) != PAPI_OK)
        exit(-1);

/* Shutdown PAPI */
PAPI_shutdown();
```

4 Hardware Implement

4.1 Platform Structures

4.1.1 Hardware

Raspberry Pi

The Raspberry Pi is a series of credit card-sized single-board computers developed in the United Kingdom by the Raspberry Pi Foundation with the intent to promote the teaching of basic computer science in schools and developing countries.

We will install our new Linux kernel and run our test programs, then measure the power consumption in RPi. Finally compare the difference of power consumption between the original kernel and our new kernel.

Figure 4.1

The following figure will show its interface:

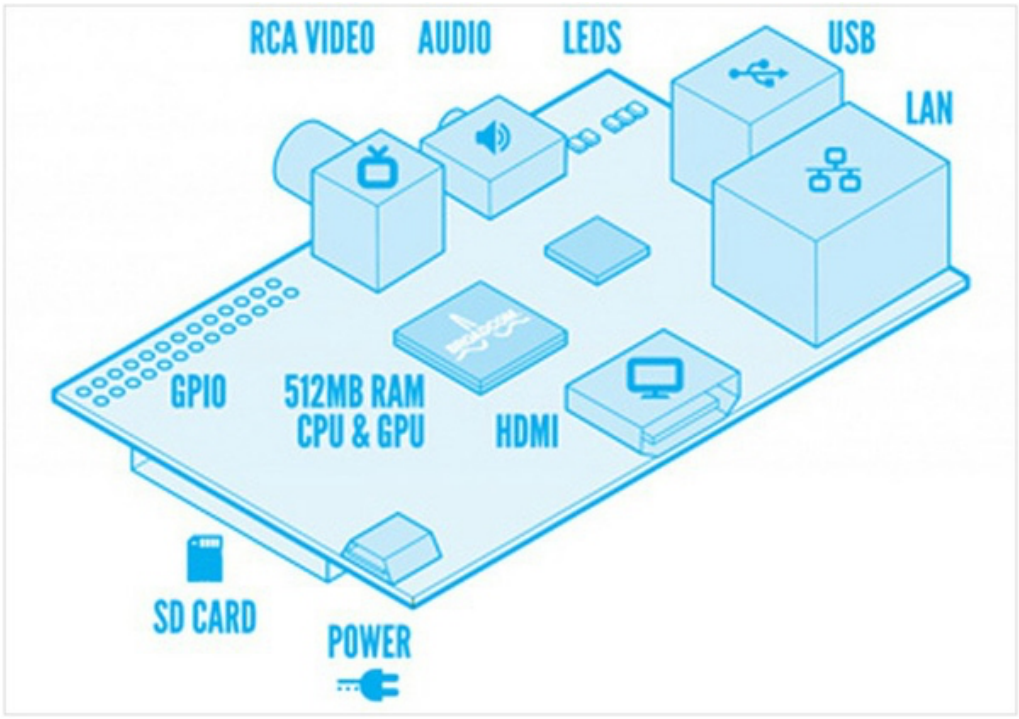

Figure 4.2

4.2 Scenario design

Following is a simple C program which can cause high cache miss.

```
1   int matrix[8192][16];   //4*8192*16=2^18=512k bytes
2
3   void bad_access()
4   {
5       int k, j, sum = 0;
6       for(k = 0; k < 16; k++)
7           for(j = 0; j < 8192; j++)
8               sum += matrix[j][k];
9   }
10
11  int main()
12  {
13      int i;
14      for(i = 0; i< 5000000; i++)
15          bad_access();
16      return 0;
17  }
```

The above code is simple, but it need to understand a simple structure and principle of cache: cache is 64 bytes or 128 bytes of a line, divided into a plurality of sets (or called multiplexing), each cache miss occurs when data is taken, cache will follow the cache line units (that is, once here take 64 bytes) of data taken from memory.

The first step is to know that the total size of the level 2 data cache is 256k, the second step tis to know each cache line is 64 bytes, therefore, level2 data cache total of 256k / 64 = 2 ^ 12 = 4096 rows.

Imagining that there is a table of 64 bytes per row, row a total of 4096, a total of 256k size, which is the simple structure of our cache. In order to ensure that each data fetch occurs miss, every time we take data, we must take the data which is larger than 64 bytes

First create an array which is 512K large and is doubled of cache. If the array is also 256k, at the end of the first cycle, the array fetch data from the beginning, the cache will not be replaced, so it will not cache miss again. So in order to cache miss every time, the array must be at least twice the size of the cache.

Reading data in the array in a loop, each time reads a size of int, then add 64, then read the next cache line data, until all the data are read up..

5 Result discussion and future work

5.1 Result discussion and next step

According to the limited time, we did not finish the part of hardware Implement and measurement of power consumption. So we cannot say that whether our algorithm is better or not, but it can really produce a creative thinking that a new way to improve our scheduling algorithm, not only the performance but also take power consumption into consideration.

There are many future extensions of this work. The most Obvious one is finish our hardware implement and measurement of power consumption, besides we should measure the cache miss rate in the kernel instead of in the user level and call a system call to feed data. Because it really causes a large overhead.

5.2 The challenges for informatics in developing software for modern multi-core computers.

Recently, multi-core computers are becoming more and more common. With the increasing number of cores and support of parallel computing, the performance and processing speed of computer improve obviously. But I have to claim that the cooperation between computer architecture and software is not an easy arithmetic like 1+1, in fact it is an overall planning and we should fuse them together. If we cannot adjust our idea of software system kernel development, even if we run the software on the most advanced multi-core computer, the performance cannot be improved and even reduced.

Before talking about what difficulties the software developer would face, I should refer to why the difficulties would appear.

First, in my opinion, one-core computer is like a small private business. It handles all the tasks and uses only a series of caches. On the other hand, multi-core computer is a large enterprise. Each core has relatively independent architecture and needs to coordinate the resource to ensure the data sharing and efficient synchronize, otherwise too much cache miss will ruin the performance.
Second, the difficulties are also caused by the difference of multi-thread technology between one-core and multi-core environment. In my operating system class, the professor told us that in the one-core computer, the multi-thread is not exactly concurrency. It is more like a round-robin and uses some hardware commands to simulate a multi-core architecture. Although it can reduce CPU idle time and improve the performance, it cannot execute many threads at the same time.

For instance, in the internship I am doing now, I also have to face the problem that how to assign the processes in one task into different cores in a heterogeneous multi-core processor. My topic is effect of process scheduling in power consumption and I decide to modify the original scheduling algorithm to make it higher energy-effective. Besides, I find a nice idea in Rajesh Patel's paper *CPU Scheduling for Power/Energy Management on Heterogeneous Multicore Processors*, which mentioned that he divided cores into several sets. One set runs at high frequency and another is at low frequency. He tried to use cache miss ratio and number of context switch as the parameter to define which process is IO bounded and which is CPU bounded. Of course IO bounded process has more priority to be assigned to the high frequency set. Besides, he synthesized the load and number of process of ready queue to decide that which core to be assigned to. After that, I focus on that in one specific core, I modify the original CFS to make CPU bounded process has more opportunity to be selected as first and has longer execution time. Although I am still working hard on it, I believe the final implement and measurement will satisfy my expectation.

According to what I have learnt from the internship, operating system and Java programming lessons, I find that nowadays, the software and kernel developer are facing the following two challenges:

1. What kind of technology can be used to optimize the existing program and how to optimize the problem of process synchronize, data competition, processor scheduling and so on?
2. Be aimed at the multi-core development environment, what kind of development pattern should be raised.

So based on the challenges I mentioned above, the following aspects should be taken into consideration when the programmers develop their software and operating system:

First, we need to make our software adapt to parallel computing and when we develop our software, we need to focus more on multi-thread. Second, for operating system, the process scheduling would be much more important. For example, we can assign the processes which need to share cache to the same core and assign the relatively memory-independent processes to the different cores. Third, the lock of protection of data access become a little bit out of date. Maybe we need to come up with a new method to adapt multi-core computer. Besides, interaction between two processes and resource preemption, in other words, how to coordinate every level cache also cannot be ignored. Last, more cores mean more power consumption. So energy saving is an everlasting problem.

Then I want to refer to some my own superficial ideas on fixing the problem in some case.

First, for software design. In my software engineering class, the professor taught us that in order to design a proper software for customers, we need to segment the whole procedure into several steps, such as software requirements, design, construction, test, maintenance and so on. So in multi-core environment, we can disintegrate the whole task into some groups. In one group, the tasks depend on each other more, but between two groups, the tasks should be more independent. For example, when we are playing a PC game, there are a lot of things we can do, such as waiting for other players, having a talk with others by a build-in chat program, crazily clicking mouse

to win the game and so no. Although one-core computer can handle this, we can sort all tasks properly and design our software to fit multi-core computer. It can obviously improve interactivity and reduce the overhead of context switch to save more energy. Similarly, for some software which handles data and computes more, we can divide the data into some groups so that improve the performance.

Second, the problem of process synchronization is thorny. When process A and B which are in two cores want to access the same data, if A goes into the critical region, B will be blocked, which means one core will be idle. It is so bad that making parallel computation like series computation. So we need to reduce the frequency of using lock or raise a new kind of synchronization mechanism.

Third, it is easy to predict that as time goes by, some cores will be too busy but some cores will be too idle. So in order to balance the load of different cores, application software developer should disintegrate their software more cautious, especially when the number of cores becomes larger. In the other hand, operating system developers should update scheduling algorithm, for example, the scheduler can check the load of core and monitor the process dynamically. Once the core is too busy, scheduler shifts some process from the core to other idle cores and scheduler should assign the process depending on the process's cache miss and context switch to avoid performance excess or deficiency.

In conclusion, the challenges in developing software mainly are 1.new software design pattern and 2. new ideas and technologies of optimize the process interaction, scheduling and data sharing.

Acknowledgements

I would like to offer a special thanks to my dissertation advisor, Professor REBAUDENGO MAURIZIO, for chairing my committee and advising me throughout my paper work. His patience, support, enthusiasm, and most importantly, his confidence in my abilities, have helped me greatly throughout my graduate study. I am also very grateful to masoud hematpour, a PHD student of Professor Rebaudengo Maurizio for his participation in my committee. This paper is dedicated to my parents and family. Their support, love, and faith in my abilities were very important for completing this paper research.