Politecnico di Torino

Academic year 2016/2017

Master degree in Electronic Engineering

Operating Systems

Report of Project

Change scheduling priority according to cache miss rate

Group composition:

Anelli Donato (231435)
Cattaneo Roberto (231205)
Marchisio Alberto (231200)

25/07/2017

1 Introduction

Starting from the previous activity work reported in "Report for internship.pdf", the purpose of this project is to verify and reproduce it and develop a testbench to test the scheduling performance.
Section 2 describes the most relevant section of the previous work, that are considered of a starting point of the activity.
Section 3 explains the system setup installed to run the experiment. It includes versions of kernel and libraries.
Section 4 shows what are the modifications in the kernel source code. In particular, which are the corrections made with respect to the previous work.
Section 5 describes the structure of the testbench adopted to make the experiment.
Finally, Section 6 discusses the results obtained.

2 Summary of previous activity

The main motivations and an example of the benefits are reported in Section 2.1, while Section 2.2 explains the idea of the implementation. Then in Section 2.3 is reported the final adopted solution.
The kernel scheduler is modified in order to take into account cache misses: processes with high cache miss rate are penalized, so they will execute less frequently. Considering the whole system, the performance is improved because the overall cache miss rate is reduced.
In addition, two new system calls are added to the table: one for inserting in the PCB the value of cache miss rate and the other for reading this data from the PCB.

2.1 Main motivations and examples

Linux 2.4

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 I/O operation. For example, a database engine is querying, it is frequently doing disk I/O. 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 realtime process.

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. It is difficult the maintenance of code: calculation of dynamic priority, average sleeping time and some obscure formula to revise priority and differentiate interactive process.

Completely fair:

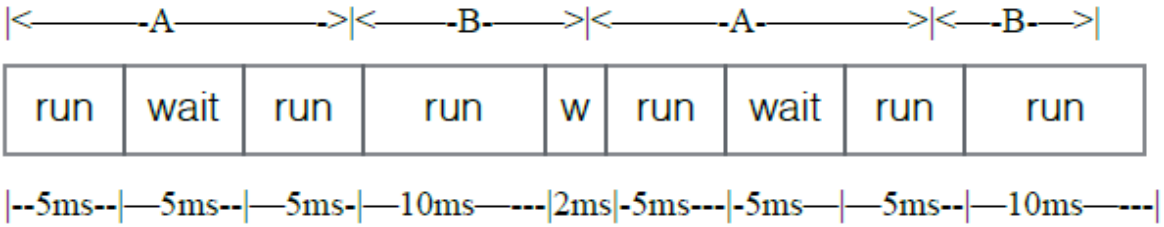
- **SD, RSDL** and **CFS** all adapt the idea of completely fair and many tests shows that the idea and CFS algorithm have very good performance. But CFS does not take power consumption into consideration. For example, there are two processes, one is a CPU bounded process A with

low priority and the other is I/O 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 I/O. 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.

Improvement

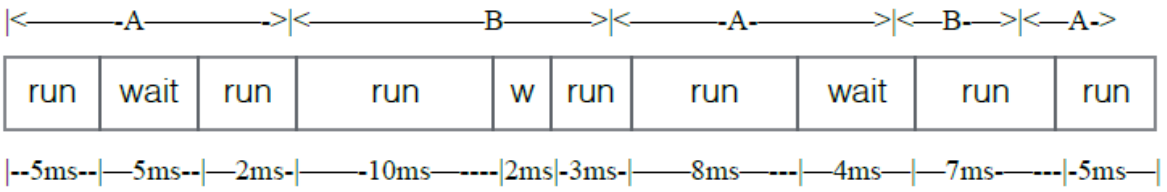
Principle:

- **For CFS:** A is I/O bounded process with high priority B is CPU bounded process with low priority



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 I/O bounded process A is I/O 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



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 I/O processes, not only the waiting time is reduced, but also the total running time will also reduce. In other words, the new CFS can save some power consumption in some case.

I/O, 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.
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 I/O 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 I/O bound process and compensate CPU bound process, in this way the chance of first picking of I/O 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.

2.2 Implementation

Ideas of implementation

Overview: first of all, 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** have been inserted. Then cache miss rate in /kernel/sched/core.c and average cache miss rate in /kernel/sched/fair.c have been initialized. Then the process will calculate its cache miss rate and feed it both into cache miss rate and average miss rate. Finally, cache miss rate and average cache miss rate is used to calculate the compensation and the punishment.

Formula:

- 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 was decided to make the system memoryless, so the formula becomes:
  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_ti me))/(number_process_queue)*

- compensate value are:
  cache_miss_rate;
  average_cache_miss_queue;
  load;

Difficulties

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

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

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

How 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.

Float data type: as we know in the linux kernel, the float data type 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.

2.3 Solution

Overview

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

- we install the PAPI library to let PAPI monitor the hardware performance. It can measure the number of L1 cache miss access in a given time
- we will write two new system calls to the system call table. One (access_pcb) help us access to process control block and the other (retrive_pcb) will retrieve the data which we feed so that can help us make sure whether our system calls work well
- instead of changing the load directly, we choose to modify the vruntime and time slice respectively with the same coefficient

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

Imitate float data type

We have
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 // percentile
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 System setup

The initial system to setup the experiment started from a clean version of Ubuntu 16.04.1 LTS. This operating system has the kernel version 4.4-generic.
Then, the original version of the kernel 4.5.3 has been downloaded and extracted. Before compiling the kernel, some modifications of the source code have been made. They are described in detail in Section 4.
Compiling the kernel with the modifications allows to use the system calls to modify the scheduling priority according to the cache miss rate.
Finally, PAPI library (version 5.5.1) has been installed in the system in order to access specific

hardware registers that count the cache misses at level 1.
It has been noticed that PAPI library works only inside systems installed directly in the hardware: virtual machines do not allow the access to some hardware registers that compute cache misses.
Moreover, this experiment works only if the program is executed by a single core: multiple core systems execute processes concurrently. If each process can be assigned to different cores and synchronization between cores is not always guaranteed. As a result, running the program with multiple cores leads to an erroneous update of cache miss rate and average.

4 Modifications of kernel source code

Most of the code modifications have been made in the previous work. However, some minor changes have been applied in order to avoid mathematical approximation errors. In "access_pcb.c" the code has been modified to apply rounding instead of truncation when converting data from floating point to integer. The code of "fair.c" has been changed to apply saturation and avoid divisions with 0 at denominator.
Overall, all the files that must be modified in the kernel source code are the following:

- /arch/x86/entry/syscalls/syscall_64.tbl: (already done in previous work) add two system calls to the table.

```
...
327     common  access_pcb               sys_access_pcb
328     common  retrive_pcb              sys_retrive_pcb
...
```

- /include/linux/sched.h: (already done in previous work) define new variables used by the scheduler.

```
...
long cache_miss_rate;
long co_1;
long co_2;
long co_3;
...
```

- /include/linux/syscalls.h: (modified in current work) add include links for the new system calls.

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

- /kernel/access_pcb.c: (modified in current work) code of the system call that updates the PCB inserting cache miss statistics.

```
#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, unsigned long Nproc){
/*'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();


        rate=nr_cachemiss/delta_time;

        if(cfs_rq){
        avg_q=cfs_rq->runnable_cachemiss_avg;
        if(rate-(se->cache_miss_rate)>=0)
        rate_q=(rate+avg_q*Nproc-(se->cache_miss_rate)+Nproc/2-1+(Nproc/2)%2)
```

```
                /Nproc;
        else
        rate_q=(rate+avg_q*Nproc-(se->cache_miss_rate)+Nproc/2+1-(Nproc/2)%2)
                /Nproc;
        }

        if(rate_q<1) rate_q=1;

        se->cache_miss_rate=rate;
        cfs_rq->runnable_cachemiss_avg=rate_q;

        put_task_struct(tsk);

        return 0;
}
```

- /kernel/Makefile: (already done in previous work) add also the two new system calls in the list.

```
...
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 access_pcb.o \
        retrive_pcb.o
...
```

- /kernel/retrive_pcb.c: (already done in previous work) read from PCB the data relative to cache miss.

```
#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=-1;
        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->co_1;

        else if(type==2)
        //retrive=se->cache_miss_nr;
        retrive=se->co_2;
        else if(type==3)
        //retrive=cfs_rq->runnable_cachemiss_sum;
        retrive=se->co_3;
        else if(type==4)
        retrive=cfs_rq->runnable_cachemiss_avg;
        else if(type==5)
        retrive=se->cache_miss_rate;
        else
        retrive=cfs_rq->nr_running;

        put_task_struct(tsk);

        return retrive;

}
```

- /kernel/sched/core.c: (already done in previous work) initialize some variables used by the scheduler.

```
...
static void __sched_fork(unsigned long clone_flags, struct task_struct *p)
{
        p->on_rq                        = 0;

        p->se.on_rq                     = 0;
        p->se.exec_start                = 0;
        p->se.sum_exec_runtime          = 0;
        p->se.prev_sum_exec_runtime     = 0;
        p->se.nr_migrations             = 0;
        p->se.vruntime                  = 0;
        p->se.cache_miss_rate           = 1;
        p->se.co_1                      = 1;
        p->se.co_2                      = 0;
        p->se.co_3                      = 0;
...
```

- /kernel/sched/fair.c: (modified in current work) change the way the scheduler updates process priority, taking into account also the cache miss rate. It is done according to the ratio between cache miss and average.

```
...
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 num;
        long den;
        long co_1;
        long co_2;
        long co_3;
        long mul;
        long acc;

        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);

        num=curr->cache_miss_rate;
        den=cfs_rq->runnable_cachemiss_avg;

        //correct den if <=0
        if(den<=0) den=1;

        co_1=num/den;
        co_2=10*num/den-co_1*10;
        co_3=100*num/den-co_1*100-co_2*10;

        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;}
        else;
```

```
        if(co_1==0){
        if(co_2<=8){
        co_2=8;co_3=0;}
        else;
        }
        else;
        curr->co_1=co_1;
        curr->co_2=co_2;
        curr->co_3=co_3;
        mul=calc_delta_fair(delta_exec, curr);
        acc=mul*co_1+(mul*co_2)/10+(mul*co_3)/100;
        curr->vruntime += acc;

        update_min_vruntime(cfs_rq);

        if (entity_is_task(curr)) {
        struct task_struct *curtask = task_of(curr);

        trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
        cpuacct_charge(curtask, delta_exec);
        account_group_exec_runtime(curtask, delta_exec);
        }

        account_cfs_rq_runtime(cfs_rq, delta_exec);
}
...
static void
check_preempt_tick(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
        unsigned long ideal_runtime, delta_exec;
        struct sched_entity *se;
        s64 delta;
        u64 slice_m;
        long co_1;
        long co_2;
        long co_3;

        long num;
        long den;
        num=cfs_rq->runnable_cachemiss_avg;
        den=curr->cache_miss_rate;

        //correct den if <=0
        if(den<=0) den=1;

        co_1=num/den;
        co_2=10*num/den-co_1*10;
        co_3=100*num/den-co_1*100-co_2*10;
```

```
/*
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;}
else;

if(co_1==0){
if(co_2<=8){
co_2=8;co_3=0;}
else;
}
else;

if((co_1!=1&&co_2!=0&&co_3!=0))
pr_emerg("Kernel panic - not syncing: Oops,");
*/

slice_m=sched_slice(cfs_rq, curr);
ideal_runtime = slice_m*co_1+(slice_m*co_2)/10+(slice_m*co_3)/100;

delta_exec = curr->sum_exec_runtime - curr->prev_sum_exec_runtime;
if (delta_exec > ideal_runtime) {
resched_curr(rq_of(cfs_rq));
/*
* The current task ran long enough, ensure it doesn't get
* re-elected due to buddy favours.
*/
clear_buddies(cfs_rq, curr);
return;
}

/*
* Ensure that a task that missed wakeup preemption by a
* narrow margin doesn't have to wait for a full slice.
* This also mitigates buddy induced latencies under load.
*/
if (delta_exec < sysctl_sched_min_granularity)
return;

se = __pick_first_entity(cfs_rq);
delta = curr->vruntime - se->vruntime;

if (delta < 0)
```

```
        return;

        if (delta > ideal_runtime)
        resched_curr(rq_of(cfs_rq));
}
...
void init_cfs_rq(struct cfs_rq *cfs_rq)
{
        cfs_rq->tasks_timeline = RB_ROOT;
        cfs_rq->min_vruntime = (u64)(-(1LL << 20));

        #ifndef CONFIG_64BIT
        cfs_rq->min_vruntime_copy = cfs_rq->min_vruntime;
        #endif
        #ifdef CONFIG_SMP
        atomic_long_set(&cfs_rq->removed_load_avg, 0);
        atomic_long_set(&cfs_rq->removed_util_avg, 0);
        #endif
        //cfs_rq->runnable_cachemiss_sum = 0;
        cfs_rq->runnable_cachemiss_avg = 1;
}
...
```

- /kernel/sched/sched.h: (already done in previous work) define new variables used by the scheduler.

```
...
long runnable_cachemiss_avg;
...
```

4.1 Update cachemiss average

The average cache miss of the system is a critical parameter. At the beginning it is initialized to 1. Also the cache miss rate of each process is initialized to 1. Then, each time a process calls access_pcb system call, the average is updated, according to the cache miss rate of the current process.
A problem arises when a process terminates its execution with a cache miss rate lower than its initial value, because it contributes to an increase of average cache miss. High average represents a problem because after a certain amount of time the value of cache miss computed by PAPI will be always higher than the average. As a consequence, the algorithm will not work properly. To avoid average divergence, before each process termination the cache miss rate must be restored to the initial value of 1. This operation is not performed inside the kernel, but it is up to the user to take care of this task.
Then the main issue is that we are dealing with integer numbers. Cache miss average is an integer number and an update requires a division. In some cases it is possible to have bad numerical approximation that lead to an erroneous average update.
In the followings, it is used the notation summarized in Table 1.

SYMBOL	NAME IN access_pcb.c	DESCRIPTION
$r(t)$	rate	Current cache miss rate of the process
$r(t-1)$	se->cache_miss_rate	Past cache miss rate of the process
$cm(t)$	nr_cachemiss	Current number of cache misses
dt	delta_time	Period of time after which the cache miss rate is updated
$ar(t)$	rate_q	Current average cache miss rate of all processes
$ar(t-1)$	avg_q	Past average cache miss rate of all processes
N	Nproc	Number of current running processes

Table 1: Symbolic notation

In the previous activity work, formulas are described in Equations (1) and (2).

$$r(t) = \frac{cm(t)}{dt} \tag{1}$$

$$ar(t) = \frac{r(t) - r(t-1) + ar(t-1) \cdot N}{N} \tag{2}$$

For a better explanation, consider an example: the main program has two children with different cache miss rate. It is desired that at the end the average cache miss rate returns to 1. In this case, computations are correct: the average returns to 1.
TIME 0: $cm_1(0) = 1;\ cm_2(0) = 1;\ ar(0) = 1;\ dt = 1$
TIME 1: $cm_1(1) = 1;\ cm_2(1) = 1$
$r_1(1) = 1;\ r_2(1) = 1$
$P1 \rightarrow ar(1) = \frac{1-1+1\cdot 3}{3} = 1 = ar(0)$
$P2 \rightarrow ar(1) = \frac{1-1+1\cdot 3}{3} = 1 = ar(0)$
So, if the cache miss rate at time t is equal to the cache miss rate at time t-1 -> average is constant.
TIME 2: $cm_1(2) = 1;\ cm_2(2) = 10$
$r_1(2) = 1;\ r_2(2) = 10$
$P1 \rightarrow ar(2) = \frac{1-1+1\cdot 3}{3} = 1 = ar(1)$
$P2 \rightarrow ar(2) = \frac{10-1+1\cdot 3}{3} = 4$
TIME 3: $cm_1(3) = 1;\ cm_2(3) = 1$
$r_1(3) = 1;\ r_2(3) = 1$
$P1 \rightarrow ar(3) = \frac{1-1+4\cdot 3}{3} = 4 = ar(2)$
$P2 \rightarrow ar(3) = \frac{1-10+4\cdot 3}{3} = 1$

Let's consider another example. In this case, at the end, the average cache miss goes to 0.
TIME 1: $cm_1(1) = 1;\ cm_2(1) = 1$
$r_1(1) = 1;\ r_2(1) = 1$
$P1 \rightarrow ar(1) = \frac{1-1+1\cdot 3}{3} = 1 = ar(0)$
$P2 \rightarrow ar(1) = \frac{1-1+1\cdot 3}{3} = 1 = ar(0)$
TIME 2: $cm_1(2) = 1;\ cm_2(2) = 20$
$r_1(2) = 1;\ r_2(2) = 20$
$P1 \rightarrow ar(2) = \frac{1-1+1\cdot 3}{3} = 1 = ar(1)$
$P2 \rightarrow ar(2) = \frac{20-1+1\cdot 3}{3} = \frac{22}{3} = 7$
TIME 3: $cm_1(3) = 1;\ cm_2(3) = 1$
$r_1(3) = 1;\ r_2(3) = 1$
$P1 \rightarrow ar(3) = \frac{1-1+7\cdot 3}{3} = 7 = ar(2)$
$P2 \rightarrow ar(3) = \frac{1-20+7\cdot 3}{3} = \frac{2}{3} = 0$
When average becomes 0 there are problems in the next cycle since, in the file kernel/fair.c there is the update_curr function, that does the following operations:

num=curr->cache_miss_rate
den=cfs_rq->runnable_cachemiss_avg
co_1=num/den
If the average cache miss (runnable_cachemiss_avg) is = 0, then there is an error (DIVISION BY 0) and the system will crash.
This issue can be fixed by applying a saturation of the average cache miss. In any case, a more precise computation of the average is desirable. This is done changing the formula inside access_pcb.c. When computing the division, it is applied rounding instead of truncation. The new formula is reported in Equations (3) and (4).

$if(r(t) - r(t-1) >= 0)$

$$ar(t) = \frac{r(t) - r(t-1) + ar(t-1) \cdot N + N/2 - 1 + (N/2)\%2}{N} \tag{3}$$

else

$$ar(t) = \frac{r(t) - r(t-1) + ar(t-1) \cdot N + N/2 + 1 - (N/2)\%2}{N} \tag{4}$$

If we apply the same example that gave at the end an average equal to 0, with the new formula the final average is 1.
TIME 1: $cm_1(1) = 1;\ cm_2(1) = 1$
$r_1(1) = 1;\ r_2(1) = 1$
$P1 \rightarrow ar(1) = \frac{1-1+1\cdot 3+3/2-1+(3/2)\%2}{3} = 1 = ar(0)$
$P2 \rightarrow ar(1) = \frac{1-1+1\cdot 3+3/2-1+(3/2)\%2}{3} = 1 = ar(0)$
TIME 2: $cm_1(2) = 1;\ cm_2(2) = 20$
$r_1(2) = 1;\ r_2(2) = 20$
$P1 \rightarrow ar(2) = \frac{1-1+1\cdot 3+3/2-1+(3/2)\%2}{3} = 1 = ar(1)$
$P2 \rightarrow ar(2) = \frac{20-1+1\cdot 3+3/2-1+(3/2)\%2}{3} = \frac{23}{3} = 7$
TIME 3: $cm_1(3) = 1;\ cm_2(3) = 1$
$r_1(3) = 1;\ r_2(3) = 1$
$P1 \rightarrow ar(3) = \frac{1-1+7\cdot 3+3/2+1-(3/2)\%2}{3} = 7 = ar(2)$
$P2 \rightarrow ar(3) = \frac{1-20+7\cdot 3+3/2+1-(3/2)\%2}{3} = \frac{3}{3} = 1$

5 Testbench

A good testbench for evaluating the change of performances must have at least two processes with different cache miss rate. For this reason, the main program creates two groups of child processes: one group should have higher cache miss rate than the other. Each process belonging to the same group performs the same function.
In order to get different cache miss rate, the two functions differ in this way: one group of processes read horizontally from a bidimensional array and the other reads from it vertically. The dimension of the array must be greater than the cache line size.
Each process computes the number of iterations for a specific amount of time. Start and end time are specified from the parent process: in this way all the processes can run concurrently.
Moreover, cache misses for each process are computed through PAPI library.
Processes can run different times according to a variable specified by the user.
Between one run and the other, the PCB of each process is modified with the value of cache miss rate computed in the previous run. In order to avoid that the average cache miss rate of the whole system will diverge, before the end of child processes' life an additional run is performed and the cache miss rate of each process is restored to the initial value.

6 Conclusions

An example of result is shown in Figure 1. It shows two executions of the same program. It has been compiled, as an example, with a 1000x1000 array, with 1 process per group (a total of two processes) and the number of runs is 2. As explained before in Section 5, at the end also a third run is executed.

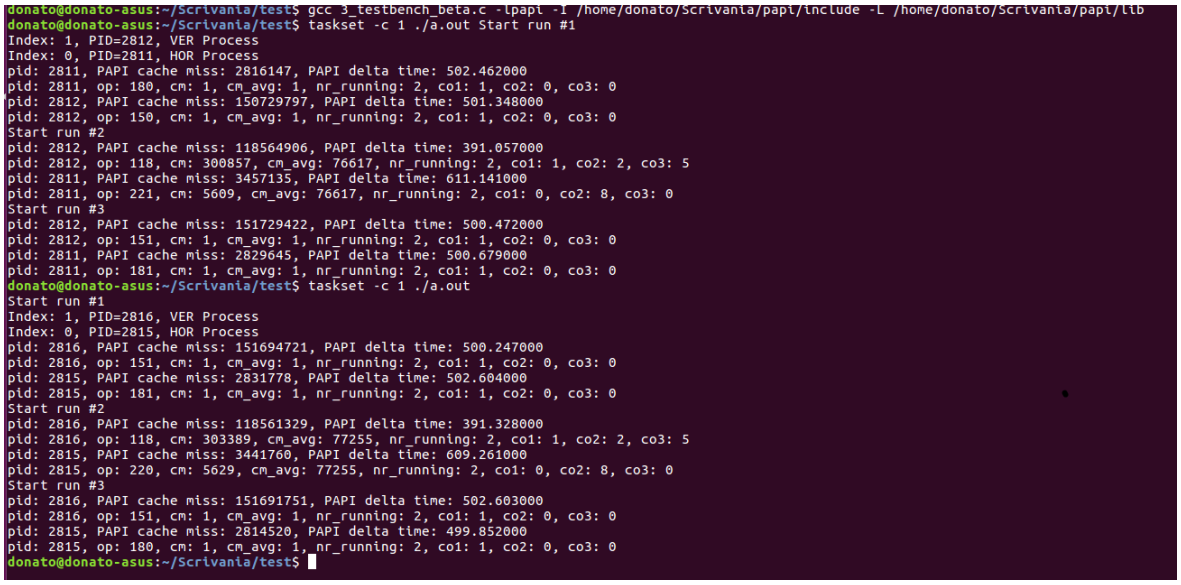


Figure 1: Output of the test program

It is evident that, at the first run, the process that reads horizontally have less cache misses than the vertical one, while the number of iterations are almost the same. At the second run, when the PCB is updated with real cache miss data, the number of iterations of each process changes consistently, according to the new scheduling policy: a process with high cache miss rate uses the CPU less frequently than another one with lower rate. At the third and last run the initial value of cache miss (equal to 1) is restored, then cache misses are similar to run 1.
Performance comparisons can be done adding together all the iterations of each process in the same run. This should be proportional to the throughput. Then computing the sum for each run means computing the performance of the system in each run. For example, consider run 1 (the first run, before inserting cache miss rate in the PCB) and run 2. Analyzing the ratio of total iterations between run 2 and run 1, an improvement of 2.7% can be achieved. This means that performances are better in run 2, because overall the total cache misses are lower. The results obtained are summarized in Table 2.

RUN	OP HOR	OP VERT	SUM OP	CM HOR	CM VERT	SUM CM.
1	180	150	330	2816147	150719797	153535944
2	221	118	339	3457135	118564906	122022041
diff			+2.7%			-20.5%

Table 2: Results comparison. OP HOR indicates the number of operations executed by the process that reads horizontally; OP VERT by the one that reads vertically; CM indicates cache misses.

Note that the testbench is parametric: results can be obtained also with more than one process per group, different number of runs and different array sizes.