Adaptive DVFS for EPOS Multicore Schedulers

Authors

- José Luis Conradi Hoffmann -> *zeluish97@gmail.com*
- Leonardo Passig Horstmann -> *leonardo.horstmann@gmail.com*

Motivation

The use of PMU to obtain data on the cores is an area that has been growing in recent years. The data extracted from the PMU are used for several types of improvement, among them we can mention as the most importants:

- Identifying regions that cause performance loss in a code due to details such as prediction error and cache faults, which in most cases, if identified can be fixed.
- Scheduling processes using the metrics obtained with the PMU, for example, on the intel website, where it's mention a performance gain with in scheduling have reached 16% {1}.

However, much still can be improved, once data that we have access in the last generation processors as temperature of the cores 2 still can be included in the heuristics used.
Tasks that require a lot processing of a core make it operate at a high frequency, which causes an increase in processor temperature. To ensure processor integrity the cooler speed increases too. In situations where this is not enough the frequency is reduced, and then the performance decreases, as can be seen in the text below, extracted from the Intel Software Developer's Manual {3} (2017, p.2944):

- If the processor's core temperature rises above the preset catastrophic shutdown temperature, the processor core halts execution, which causes both logical processors to stop execution.
- When the processor's core temperature rises above the preset automatic thermal monitor trip temperature, the frequency of the processor core is automatically modulated, which effects the execution speed of both logical processors.

With this informations, it is possible to manage the temperature in order to reduce the energy consumption of the processor, through reduction processor's frequency and voltage.

Goals

Use temperature sensors in the EPOS PMU to configure DVFS (Dynamic Voltage and Frequency Scaling) to run in order to decrease the frequency (and voltage) achieving so a reduction in temperature and consumption of the system.
Trying to avoid too many interrupts checking the DVFS activation conditions, we will use the cpu "free time" (Thread Idle) to perform most of the checks, instead of only PMU Interruptions. Yet, some PMIs will still occur during execution, but less frequently.

Methodology

We will use for the accomplishment of this work a set of methods which can be described by:

- **Studying**: Consists of reading in order to solve doubts, and to execute tests on existing codes (Simulate), so the appropriate knowledge of the tool and characteristics of the current "model" can be obtained.
- **Code**: It consists of making new code or correcting and/or adapting existing code from the knowledge gained and decisions made during the studies.

- **Testing**: Test the codes implemented, corrected and/or adapted in order to verify correctness and characteristics of the proposed algorithms.
- **Report**: Take note of studies, implementations and tests performed in order to obtain a basis for guiding the following processes.

Tasks

1. Detailed Project Plan.
2. List the tools (software) that will be used and test them in order to ensure their operation and knowledge of their specifications.
3. Study documentation and test operation and data capture of EPOS PMU and PMIs.
4. Implement and test reading statistics from MSRs using PMU interruptions and Thread Idle, that in the future will activate the frequency and voltage scaling function (DVFS).
5. Study and test when to activate DVFS based on thermal sensors reading of the processors, PMU status and deadline misses
6. Implement necessary checks to call the DFVS, compare results obtained and submit final report.

Deliverables

1. Detailed Project Plan, Report of the study contemplating realized researches and results found.
2. Demonstration of Technological Viability.
3. Report of tests performed and knowledge acquired on PMU, PEBS and PMI.
4. Source code, tests and results.
5. Report of tests performed and knowledge acquired on triggering DVFS.
6. Project source code and final report with conclusions obtained.

Schedule

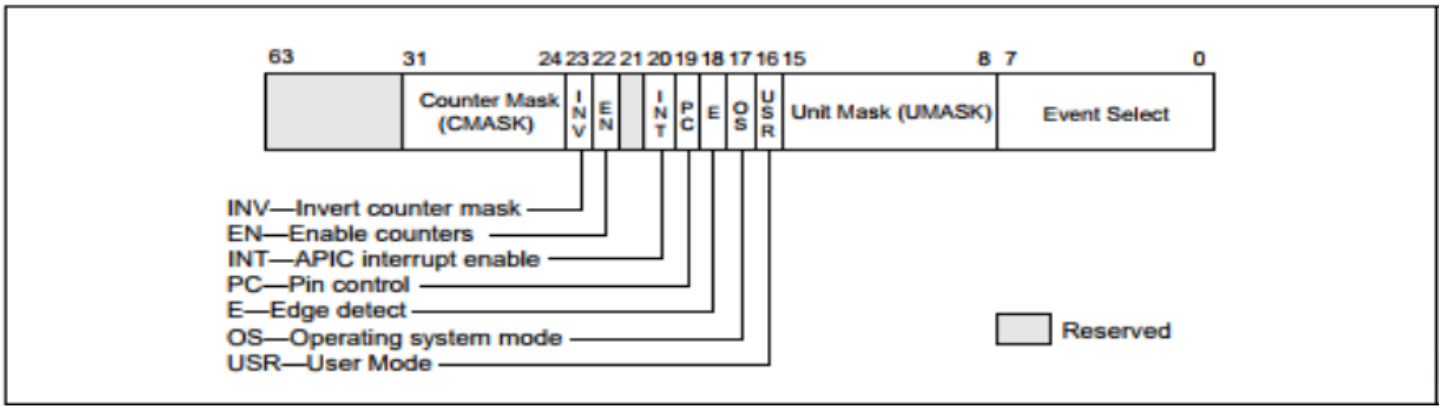
Task	25/09	02/10	11/10	16/10	25/10	01/11	06/11	15/11	29/11
Task1	D1								
Task2	X	D2							
Task3		X	X	D3					
Task4				X	X	X	D4		
Task5							X	D5	
Task6								X	D6

Important Definitions

PMU

PMU can be defined as a set of MSR's (model-specific performance-monitoring counter), which allows the selection of parameters, to be monitored and measured, of processor performance.
The information obtained by the counters (PMC) can be used for many types of optimization, like performance, energy and even compilation.

MSR’s Layout:



Events on ID V1:

Table 18-1. UMask and Event Select Encodings for Pre-Defined Architectural Performance Events

Bit Position CPUID.AH.EBX	Event Name	UMask	Event Select
0	UnHalted Core Cycles	00H	3CH
1	Instruction Retired	00H	C0H
2	UnHalted Reference Cycles	01H	3CH
3	LLC Reference	4FH	2EH
4	LLC Misses	41H	2EH
5	Branch Instruction Retired	00H	C4H
6	Branch Misses Retired	00H	C5H

Over time, Intel made several improvements (from V1 to V4), the most important are:

- Addition of fixed registers (CTR0 to CTR2) for a faster and simpler access to the most frequently used events.
    - CTR0: Instructions Retired from V1.
    - CTR1: UnHalted Core Cycles and UnHalted Thread Cycles from V1.
    - CTR2: UnHalted Referenced Cycles from V1 using TSC.
    - CTR_CTRL: General control for the fixed registers.

- Adding registers for status and global settings to speed up access to settings and general information.
    - IA32_PERF_GLOBAL_CTRL: enable or disable counting.
    - IA32_PERF_GLOBAL_STATUS: check counter overflow conditions.
    - IA32_PERF_GLOBAL_OVF_CTRL: clear the counter overflow conditions.

- InUse bits for each register (in a specific register), thus allowing more than one thread use the PMU system simultaneously, limited only by the amount of the MSR's present, without one affecting the other analysis.

- IA32_PERFEVTSELx Bit 21 (AnyThread): Provides configuration of data analysis of logical cores together or just one of them. And in IA32_PERF_FIXED_CTR_CTRL for the fixed ones.

- Global registers to Set and Reset.

- PMI: Interruptions are generated by an MSR configured to do so or by interrupts generated by the

PEBS, freezing LBR information and/or counters in order to remain them consistent until the end of the handler routine.

PEBS

PEBS is a feature available on Intel processors that allows sampling of the current state of the machine to be stored in pre-designated memory spaces. PEBS can be understood, in a simpler way, also as a special counting mode, in which counters can be configured so that, from its overflow, the processor is interrupted and the state of the machine is recorded.

In general, any of the general purpose registers can be used for PEBS, as long as the performance event supports PEBS. The software uses IA32_MISC_ENABLE 7 and IA32_MISC_ENABLE 12 to detect whether the performance monitoring mechanism and PEBS functionality are supported in the processor. The IA32_PEBS_ENABLE MSR provides 4 bits that the software must use to enable which IA32_PMCx overflow condition will cause the PEBS record to be captured.

Once PEBS is configured, when the selected counter overflows, PEBS has its hardware armed, and in the next event occurrence a assist, which causes writing of a PEBS record, is triggered. Once an instruction causes a PEBS event to be written, the return instruction pointer (RIP) will point to the first instruction after the instruction which has caused the activation.

A valid observation at this point is that not all performance events are supported in PEBS, only a small subset of these is supported, and this set may vary depending on the working version.

DVFS

Dynamic Voltage and Frequency Scaling (DVFS, also called DVS - Dynamic Voltage Scaling) is a technique introduced in the 1990s to reduce power consumption in digital systems by reducing its voltage and frequency of operation according to its workload.

DVFS allows system to change the supplied voltage in real-time, without requiring to reset the computer, and can be used both to reduce power consumption or to achieve best performance.

According to {7}, the dependence of the energy cost of a computation on the processor core voltage and frequency is a complex function of system configuration and properties of the application. What makes to predict an energy-optimal operating point for DVS too much complex using simple models.

Having said this, we can assume that {7} in general determine optimal operation is only possible if application loads are known well in advance, so on, the only alternative is to determine the optimal voltage and frequency setting at run-time, based on the observation of the actual power consumption.

CEDF

Clustered Earliest Deadline First (multi-core), extends EDF scheduling.
Code from EPOS CEDF:

- // QUEUES x HEADS must be equal to Traits<Machine>::CPUS
- static const unsigned int HEADS = 2;
- static const unsigned int QUEUES = Traits<Machine>::CPUS / HEADS;

According to EPOS code above, we can verify that with two heads and the number of queues being (CPUS /

HEADS), the scheduler uses a queue for each physical core and each of these queues has two heads, that is, one for of each logical core, thus covering all the logical cores present in the machine.

Making a clustering by this way, the cores are clustered according to their proximity, in others words, it clusters the cores that share a common cache. Then the scheduling can have a much lower cost and synchronization time, being a multi-core scheduling.

It can be aperiodic or periodic (one constructor for each type).
Define:

- static unsigned int current_queue() { return Machine::cpu_id() / HEADS; }
- static unsigned int current_head() { return Machine::cpu_id() % HEADS; }

In short, it's an EDF with multiple two-headed queue.

Thread Idle

We can use CPU "free time" to verify conditions to activate the DVFS.

Here is the code of what idle do, you can find it on "src/component/Thread.cc" on EPOS source code.

```
int Thread::idle()
{
    while(_thread_count > Machine::n_cpus()) { // someone else besides idles
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(CPU=" << Machine::cpu_id() << ",this=" <<
running() << ")" << endl;
        CPU::int_enable();
        CPU::halt();
        if(_scheduler.schedulables() > 0) // A thread might have been woken up by another
CPU
            yield();
    }

    CPU::int_disable();
    if(Machine::cpu_id() == 0) {
        db<Thread>(WRN) << "The last thread has exited!" << endl;
        if(reboot) {
            db<Thread>(WRN) << "Rebooting the machine ..." << endl;
            Machine::reboot();
        } else
            db<Thread>(WRN) << "Halting the machine ..." << endl;
    }
    CPU::halt();

    return 0;
}
```

Demonstration of Technological Viability

The task's focus was to test parts of the system and also some tools that can be to be used on the project development.

We conducted tests to show that it is possible to use EPOS resources to develop the project and achieve the

proposed goals.

PMU, PEBS and PMI

The guiding teachers of the project have actively participated in the development of this part of EPOS, so the support for the part of the development that uses these functions is more accessible and its operation and maintenance has greater guarantee, therefore the risks involving these elements are smaller.

Thermal Sensors

According to the intel manual {9}, in the IA32 architecture, a processor temperature can be read using the IA32_THERM_STATUS MSR.

According to the manual, in the topic "14.7.5.2 Reading the Digital Sensor" that can be found in the "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2" {9} (2016, p.27), we find the following way of reading the temperature value:

- Unlike traditional analog thermal devices, the output of the digital thermal sensor is a temperature relative to the maximum supported operating temperature of the processor.
- Digital Readout (bits 22:16, RO) — Digital temperature reading in 1 degree Celsius relative to the TCC activation temperature.
    - 0: TCC Activation temperature,
    - 1: (TCC Activation - 1) , etc. See the processor's data sheet for details regarding TCC activation.
    - A lower reading in the Digital Readout field (bits 22:16) indicates a higher actual temperature.

The IA32_THERM_STATUS MSR's address is 0x19CH, what can be verified on the following table, whose was take from the same manual:

Table 14-1. Architectural and Non-Architectural MSRs Related to HWP

Address	Architectural	Register Name	Description
770H	Y	IA32_PM_ENABLE	Enable/Disable HWP.
771H	Y	IA32_HWP_CAPABILITIES	Enumerates the HWP performance range (static and dynamic).
772H	Y	IA32_HWP_REQUEST_PKG	Conveys OSPM's control hints (Min, Max, Activity Window, Energy Performance Preference, Desired) for all logical processor in the physical package.
773H	Y	IA32_HWP_INTERRUPT	Controls HWP native interrupt generation (Guaranteed Performance changes, excursions).
774H	Y	IA32_HWP_REQUEST	Conveys OSPM's control hints (Min, Max, Activity Window, Energy Performance Preference, Desired) for a single logical processor.
777H	Y	IA32_HWP_STATUS	Status bits indicating changes to Guaranteed Performance and excursions to Minimum Performance.
19CH	Y	IA32_THERM_STATUS[bits 15:12]	Conveys reasons for performance excursions
64EH	N	MSR_PPERF	Productive Performance Count.

Intel defines a certain Tjunction temperature for the processor. This value is usually in the range between 85ºC and 105ºC. In the later generation of processors, starting with Nehalem, the exact. Tjunction Max value is available for software to read in an MSR (IA32_TEMPERATURE_TARGET -> Address 0x1a2). So the actual temperature is calculated like this 'Core Temp = Tjunction - Delta'.

C code test

For a primal test of the functions, we tested a c code to run on a Linux Ubuntu. We tried to run as a simple user and as root, both tests ended in the same error: *Segmentation fault (core dumped).*

The c code tested is:

```
#include<stdio.h>
#include<stdlib.h>

int main() {
        unsigned long therm_status_address = 0x19c;
        unsigned long temp_target_address = 0x1a2;
        unsigned long long therm_read;
        unsigned long long target_read;
        asm volatile ("rdmsr": "=A"(therm_read): "c"(therm_status_address));
        printf("msr read %llu \n", (therm_read));
        //int bits = 22 - 16 + 1;
        therm_read >>= 16;
        therm_read &= (1ULL << (7)) - 1;
        asm volatile ("rdmsr": "=A"(target_read): "c"(temp_target_address));
        printf("msr read %llu \n", (target_read));
        //bits = 23 - 16 + 1;
        target_read >>= 16;
        target_read &= (1ULL << (8)) - 1;
        printf("msr read %llu \n", (target_read - therm_read));
        return 0;
}
```

Analyzing the kernel log on /var/log/kern.log we can see exactly what happens:

```
Oct  9 16:22:24 leonardo-Inspiron-5537 kernel: [ 5389.734913] traps: read[19902] general
protection ip:400545 sp:7ffea67ba550 error:0 in read[400000+1000]
```

Kernel Module test

Reading more about the reported error and the MSR's read operation we concluded that these registers can only be read on SO (Kernel) mode. So, to read the MSR's and get the CPU temperature we created a kernel module {14}, the code is placed bellow:

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

int init_module(void)
{
        unsigned long therm_status_address = 0x19c;
        unsigned long temp_target_address = 0x1a2;
        unsigned long long therm_read;
        unsigned long long target_read;
        asm volatile ("rdmsr": "=A"(therm_read): "c"(therm_status_address));
        printk(KERN_INFO "msr read %llu \n", (therm_read));
        //int bits = 22 - 16 + 1;
        therm_read >>= 16;
        therm_read &= (1ULL << (7)) - 1;
        asm volatile ("rdmsr": "=A"(target_read): "c"(temp_target_address));
        printk(KERN_INFO "msr read %llu \n", (target_read));
        //bits = 23 - 16 + 1;
        target_read >>= 16;
        target_read &= (1ULL << (8)) - 1;
        printk(KERN_INFO "msr read %llu \n", (target_read - therm_read));
```

```
        return 0;
}

void cleanup_module(void)
{
        unsigned long therm_status_address = 0x19c;
        unsigned long temp_target_address = 0x1a2;
        unsigned long long therm_read;
        unsigned long long target_read;
        asm volatile ("rdmsr": "=A"(therm_read): "c"(therm_status_address));
        printk(KERN_INFO "msr read %llu \n", (therm_read));
        //int bits = 22 - 16 + 1;
        therm_read >>= 16;
        therm_read &= (1ULL << (7)) - 1;
        asm volatile ("rdmsr": "=A"(target_read): "c"(temp_target_address));
        printk(KERN_INFO "msr read %llu \n", (target_read));
        //bits = 23 - 16 + 1;
        target_read >>= 16;
        target_read &= (1ULL << (8)) - 1;
        printk(KERN_INFO "msr read %llu \n", (target_read - therm_read));
        printk(KERN_INFO "Goodbye world 1.\n");
}
```

Init_module is the function executed when module is added to kernel and cleanup_module is the function executed on module removal from kernel. So, in this code, we are reading the temperature two times.
To compile the code, we used a makefile available on {14}. To put the code on kernel mode we used the insmod call and to remove it from kernel modules we used rmmod. The makefile code and the calls are (the name of our file was read.c):

Makefile:

```
obj-m += read.o

all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

insmod command:

```
insmod read.ko
```

rmmod command:

```
rmmod read
```

Module execution results:

After execute rmmod command we can see the result of the reads on kern.log:

```
Oct  9 16:24:56 leonardo-Inspiron-5537 kernel: [ 5541.195104] msr read 2285242376
Oct  9 16:24:56 leonardo-Inspiron-5537 kernel: [ 5541.195106] msr read 40108032
Oct  9 16:24:56 leonardo-Inspiron-5537 kernel: [ 5541.195107] msr read 46
Oct  9 16:25:05 leonardo-Inspiron-5537 kernel: [ 5550.293759] msr read 2285438984
Oct  9 16:25:05 leonardo-Inspiron-5537 kernel: [ 5550.293763] msr read 40108032
Oct  9 16:25:05 leonardo-Inspiron-5537 kernel: [ 5550.293764] msr read 43
```

```
Oct  9 16:25:05 leonardo-Inspiron-5537 kernel: [ 5550.293765] Goodbye world 1.
```

Remote test

Using remote boot on a real machine of LISHA (Software/Hardware Integration Lab - UFSC), 150.162.217.58:5000, we tested the following code, placed on the main() of an EPOS application code, to read the necessary registers and calculate the temperature:

```
using namespace EPOS;

OStream cout;
int main()
{
     //register declaration
    CPU::Reg64 ret;
    CPU::Reg32 thermalStatus = 0x19c;
    CPU::Reg64 delta;
    //reading msr in assembly
    asm volatile ("rdmsr": "=A"(delta): "c"(thermalStatus));
    //number of bits needed
    int bits = 22 - 16 + 1;
    //Show only part of the register
    delta >>= 16;
    delta &= (1ULL << bits) - 1;
    //print the delta[22:16]
    cout<<delta<<endl;
     //register declaration
    CPU::Reg32 temperatureTarget = 0x1a2;
    CPU::Reg64 tJoint;

    //reading msr in assembly
    asm volatile("rdmsr": "=A"(tJoint): "c"(temperatureTarget));
    //number of bits needed
    bits = 23 - 16 + 1;
    //Show only part of the register
    tJoint >>= 16;
    tJoint &= (1ULL << bits) - 1;
    //print tJoint[23:16]
    cout<<tJoint<<endl;
    long long d1 = (long long) delta;
    long long tJ1 = (long long) tJoint;
    //temperature = IA32_TEMPERATURE_TARGET[23:16] - IA32_THERM_STATUS[22:16]
    long long temp = tJ1 - d1;
    //print the result
    cout<<temp<<endl;
    //testing with CPU::rdmsr() from EPOS
    ret = CPU::rdmsr(0x19c);
    //print the result
    cout<<hex<<ret<<endl;
    return 0;
}
```

Remote execution result:

The execution result was:

```
<0>: PCI: device [0:2.0] reports implausible large region. Ignoring! :<0>
Setting up this machine as follows:
  Processor:    8 x IA32 at 3392 MHz (BUS clock = 12 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261700 Kbytes [0x00000000:0x0ff91000]
  PCI aperture: 5142 Kbytes [0xfe000000:0xfe505800]
  Node Id:      will get from the network!
  Setup:        22624 bytes
  APP code:     20560 bytes     data: 576 bytes
56
98
42
0x88380000
<0>: The last thread has exited! :<0>
<0>: Rebooting the machine ... :<0>
\00
```

This result was printed on a log file that is posted on the line that represents the execution of the uploaded file.
The value to be taken from the last print is on “38” (bits 16 to 22) that converting to decimal is 3*16 + 8 = 56, the same value of the first print, exactly what was expected (to be the same value).

Clock Modulation (Frequency)

According to Intel sdm {3} "14.7.3 Software Controlled Clock Modulation", a way to implement energy management is to use Software Controlled Clock Modulation, where is possible to change the processor duty cycle using the IA32_CLOCK_MODULATION MSR, and by this, reduce power consumption.

- On-Demand Clock Modulation Enable, bit 4 — Enables on-demand software controlled clock modulation when set; disables software-controlled clock modulation when clear.
- On-Demand Clock Modulation Duty Cycle, bits 1 through 3 — Selects the on-demand clock modulation duty cycle (see Table 14-3). This field is only active when the on-demand clock modulation enable flag is set.

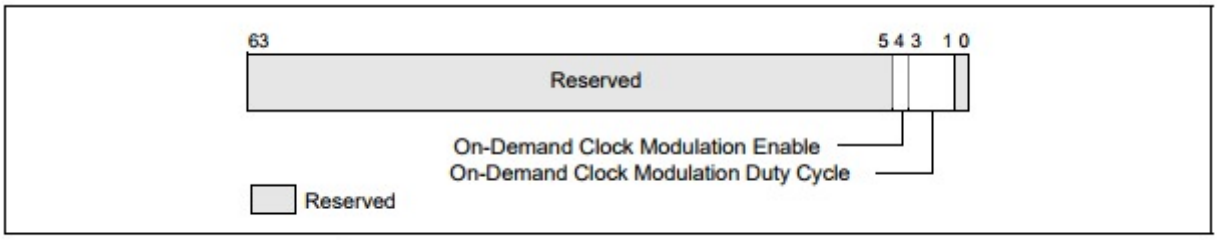


Figure 14-25. IA32_CLOCK_MODULATION MSR

For multiple processor cores in a physical package, each processor core can modulate to a programmed duty cycle independently.

Table 14-3. On-Demand Clock Modulation Duty Cycle Field Encoding

Duty Cycle Field Encoding	Duty Cycle
000B	Reserved
001B	12.5% (Default)
010B	25.0%
011B	37.5%
100B	50.0%
101B	63.5%
110B	75%
111B	87.5%

Used Code

```
unsigned long IA32_CLOCK_MODULATION = 0x19a;
//formating... 0...10010 -> B is reserved -> bit 4 to active
//1 in the second bit -> 1B -> 12.5% (default)
unsigned long long modulation = 0x12;
asm volatile ("wrmsr": "=A"(modulation): "c"(IA32_CLOCK_MODULATION));
```

PMU, PEBS and PMI on EPOS

PMU

EPOS has a file named pmu.h in "*include/pmu.h* (*EPOS PMU Mediator Common Package*)", where we can find some settings that are on both Intel PMU and Arm PMU.
Intel version of PMU can be found in "*include/architecture/ia32/pmu.h*". In this file we find the following definitions:

- Intel_PMU_V1
- Intel_PMU_V2
- class Intel_Core_Duo_PMU: public Intel_PMU_V2
- class Intel_Core_Micro_PMU: public Intel_PMU_V2
- Definition of Intel_PMU_V3 e class Intel_Sandy_Bridge_PMU: public Intel_PMU_V3

Attended events in each version follows Intel standard, to check if an event is available or not, just check if the event is available in the manual version of intel {3}.

Test

We have executed some tests that are available in EPOS named *pmu_test.cc*, what can be found in "*src/architecture/ia32/pmu_test.cc*" and "*src/architecture/ia32/pmu_test_traits.cc*". We ran it using remote boot on a real machine of LISHA (Software/Hardware Integration Lab - UFSC), 150.162.217.58:5000.

The code configures and reads some events on the PMU. They were read (and printed) 5 times, in each of these times they are printed in three different conditions: after start (*PMU: : start(i)*), after stop (*PMU: : stop(i)*) and after reset (*PMU: :reset(i)*).

Tested Code

```
PMU::config(0, PMU::INSTRUCTION); //first parameter is channel, second is the event
PMU::config(1, PMU::DVS_CLOCK);
```

```
PMU::config(2, PMU::CLOCK);
PMU::config(3, PMU::CACHE_HIT);
PMU::config(4, PMU::BRANCH);
for(unsigned int j = 0; j < 5; j++) {
   for(unsigned int i = 0; i < PMU::CHANNELS; i++)
       PMU::start(i);
   for(unsigned int i = 0; i < PMU::CHANNELS; i++)
       cout << "PMU::Counter[" << i << "]=" << PMU::read(i) << endl;
   for(unsigned int i = 0; i < PMU::CHANNELS; i++)
       PMU::stop(i);
   for(unsigned int i = 0; i < PMU::CHANNELS; i++)
       cout << "PMU::Counter[" << i << "]=" << PMU::read(i) << endl;
   for(unsigned int i = 0; i < PMU::CHANNELS; i++)
       PMU::reset(i);
   for(unsigned int i = 0; i < PMU::CHANNELS; i++)
       cout << "PMU::Counter[" << i << "]=" << PMU::read(i) << endl;
}
```

PEBS

On the version we work of EPOS, (https://epos.lisha.ufsc.br/svn/epos2/trunk/) there is no PEBS implementation.

PMI

According to Intel Software Developer Manual {3}, PMI is like any other simple interruption, so it is supported from EPOS, like an overflow (exactly what it is).

Thermal Class

We have created a class to read the temperature from the CPU MSRs. It uses the rdmsr() CPU function on the CPU that is currently running.

```
#ifndef __ia32_thermal_h
#define __ia32_thermal_h

#include <cpu.h>

__BEGIN_SYS

class Thermal
{

public:
    Thermal() {}

    static unsigned int read_temperature() {
                CPU::Reg32 IA32_THERM_STATUS = 0x19c;
                CPU::Reg32 IA32_TEMPERATURE_TARGET = 0x1a2;
                CPU::Reg64 therm_read = CPU::rdmsr(IA32_THERM_STATUS);
        CPU::Reg64 temp_target_read = CPU::rdmsr(IA32_TEMPERATURE_TARGET);
                int bits = 22 - 16 + 1;
                therm_read >>= 16;
```

```
                therm_read &= (1ULL << bits) - 1;
                bits = 23 - 16 + 1;
                temp_target_read >>= 16;
                temp_target_read &= (1ULL << bits) - 1;
                unsigned int temp = (unsigned int)(temp_target_read - therm_read);
                return temp;
    }
};

__END_SYS

#endif
```

Statistics Collecting

PMU and PMI

The first executed test was using a PMU interruption (PMI). PMU was configured on 2 events, Level 3 Cache Misses and Retired Instructions, using a pre-configured handler that do only a print of the temperature (the temperature read corresponds to the temperature of the CPU currently running)

```
// EPOS Synchronizer Component Test Program

#include <utility/ostream.h>
#include <pmu.h>
#include <utility/handler.h>
#include <chronometer.h>
#include <thread.h>
#include <display.h>
#include <utility/random.h>
#include <clock.h>
#include <mmu.h>
#include <semaphore.h>
#include <clock.h>
#include <architecture/ia32/thermal.h>
//#include <perf_mon.h>

using namespace EPOS;

OStream cout;

#define CHANNEL 6
#define CHANNEL2 5

void int_handler() {
    cout << "HANDLER -> THERMAL STATUS: " << Thermal::temperature() << endl;
}



int teste2(){
                //Disable interrupt
        APIC::disable_perf_int();
```

```
        //stop counter
        PMU::stop(CHANNEL);
        PMU::stop(CHANNEL2);
        // Set PMC Start Value
        PMU::write(CHANNEL, -1000);
        PMU::write(CHANNEL2, -1000);
        // set interrupt handler
        PMU::handler(int_handler, CHANNEL);
        PMU::handler(int_handler, CHANNEL2);
        // Set PMC Event and Start Counter
        PMU::config(CHANNEL, PMU_Common::INSTRUCTION, PMU::INT);
        PMU::config(CHANNEL2, PMU_Common::L3_MISS, PMU::INT);
        // Enable interrupt
        PMU::start(CHANNEL);
        PMU::start(CHANNEL2);
        APIC::enable_perf_int();


    int vetor[1000];

    for (int i=0;i<300;i++){
        vetor[i] = vetor[i] +1 ;
        // cout << PMU::read(CHANNEL) << ":" << Machine::cpu_id() << ":" << PMU::overflow()
<< endl;
        // cout << PMU::read(CHANNEL2) << ":" << Machine::cpu_id() << ":" << PMU::overflow()
<< endl;
}

cout << "thread terminada" << endl;

return 0;

}


int main()
{

    Thread* cons[8];

        for(int i=0;i<8;i++){
            Thread::Configuration   conf(Thread::READY,
                            Thread::Criterion(Thread::NORMAL,i)
                            );
            cons[i] = new Thread(conf, &teste2);
            cons[i]->join();
        }


    cout << "The end!" << endl;


    return 0;
}
```

Result

```
<0>: PCI: device [0:2.0] reports implausible large region. Ignoring! :<0>
Setting up this machine as follows:
  Processor:    8 x IA32 at 3392 MHz (BUS clock = 12 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261700 Kbytes [0x00000000:0x0ff91000]
  PCI aperture: 5142 Kbytes [0xfe000000:0xfe505800]
  Node Id:      will get from the network!
  Setup:        22624 bytes
  APP code:     24832 bytes     data: 608 bytes
thread terminada
threHANDLER -> THERMAL STATUS: 52
ad terminada
threadHANDLER -> THERMAL STATUS: 51
 terminada
threadHANDLER -> THERMAL STATUS: 51
 terminada
threadHANDLER -> THERMAL STATUS: 47
 terminada
threadHANDLER -> THERMAL STATUS: 48
 terminada
threadHANDLER -> THERMAL STATUS: 48
 terminada
threadHANDLER -> THERMAL STATUS: 49
 terminada
The end!
<0>: The last thread has exited! :<0>
<0>: Rebooting the machine ... :<0>
```

Thread Idle

Here we use almost the same code that runs inside PMU handler presented above.

```
int Thread::idle()
{
    while(_thread_count > Machine::n_cpus()) { // someone else besides idles
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(CPU=" << Machine::cpu_id() << ",this=" <<
running() << ")" << endl;
        //this is the inserted code
        if (Machine::cpu_id() == 0) {
                db<Thread>(WRN) << "THERMAL_STATUS: "<< Thermal::temperature()<<endl;
        }
        //end of inserted code
        CPU::int_enable();
        CPU::halt();
        if(_scheduler.schedulables() > 0) // A thread might have been woken up by another
CPU
            yield();
    }

    CPU::int_disable();
    if(Machine::cpu_id() == 0) {
```

```
        db<Thread>(WRN) << "The last thread has exited!" << endl;
        if(reboot) {
            db<Thread>(WRN) << "Rebooting the machine ..."<< running() << endl;
            Machine::reboot();
        } else
            db<Thread>(WRN) << "Halting the machine ..." << endl;
    }
    CPU::halt();
    return 0;
}
```

This is used to check the temperature in the CPU "free time".

Resultant Log

Here we show the machine configuration and some of the thermal reads.

```
<0>: PCI: device [0:2.0] reports implausible large region. Ignoring! :<0>
Setting up this machine as follows:
  Processor:    8 x IA32 at 3392 MHz (BUS clock = 12 MHz)
  Memory:       262144 Kbytes [0x00000000:0x10000000]
  User memory:  261700 Kbytes [0x00000000:0x0ff91000]
  PCI aperture: 5142 Kbytes [0xfe000000:0xfe505800]
  Node Id:      will get from the network!
  Setup:        22624 bytes
  APP code:     36944 bytes     data: 704 bytes
...
<0>: THERMAL_STATUS: 48 :<0>
<0>: THERMAL_STATUS: 47 :<0>
<0>: THERMAL_STATUS: 47 :<0>
<0>: THERMAL_STATUS: 48 :<0>
<0>: THERMAL_STATUS: 47 :<0>
<0>: THERMAL_STATUS: 48 :<0>
...
```

Temperature Storage Struct

To keep the read temperature and avoid unnecessary reads, we used a vector to store the read values of the cores's temperature on the position indicated by it's cpu number.

Thread.h code:

```
protected:
      static int _cpu_temperature[Traits<Build>::CPUS];
```

This vector is initialized on Thread_init code with the PMU initialization (described on next topic).

PMU Initialization

To start PMU data collecting we created, first, a conditional attribute on Creterion (on Scheduler.h class Priority), where we use a energy_aware attribute as the following code:

```
static const bool energy_aware = true;
```

Finally, to configure PMU, we used the code on Thread_init.

```
//Inicialization of PMU Channels to the statistics collecting in energy aware policies
    // Capturing INSTRUCTION, LLC_MISS, DVS_CLOCK.
    if(Criterion::energy_aware) {
        _cpu_temperature[Machine::cpu_id()] = Thermal::temperature();
        APIC::disable_perf_int();

        PMU::stop(3);
        PMU::stop(4);
        PMU::stop(5);
        PMU::reset(3);
        PMU::reset(4);
        PMU::reset(5);

        PMU::write(3, 0);
        PMU::write(4, 0);
        PMU::write(5, 0);

        PMU::config(3, PMU::INSTRUCTION);
        PMU::config(4, PMU::LLC_MISS);
        PMU::config(5, PMU::DVS_CLOCK);
        PMU::start(3);
        PMU::start(4);
        PMU::start(5);
        APIC::enable_perf_int();
    }
```

The channels 1 and 2 of PMU are reserved, so we decided to start counting on 3.

Thermal statics collecting

The methods used to collect the staticst were chosen because they do not break the normal execution flow, avoiding interrupts and jitter.

To collect the thermal statistics we used the following code on Thread: :dispatch():

```
void Thread::dispatch(Thread  prev, Thread  next, bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();

        //ADVFS
        if(Criterion::energy_aware) {
            _cpu_temperature[Machine::cpu_id()] = Thermal::temperature();
        }
    }
   ...
}
```

We also used another Thread code to read the cores's temperature, the thread idle code is below:

```
int Thread::idle()
{
    while(_thread_count > Machine::n_cpus()) { // someone else besides idles
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(CPU=" << Machine::cpu_id() << ",this=" <<
running() << ")" << endl;
        if(Criterion::energy_aware) {
            _cpu_temperature[Machine::cpu_id()] = Thermal::temperature();
        }
        CPU::int_enable();
        CPU::halt();
        if(_scheduler.schedulables() > 0) // A thread might have been woken up by another
CPU
            yield();
    }
...
}
```

Frequency Change Operation and Impact Measurement

Based on the IA32_CLOCK_MODULATION MSR, explained above on this page, we use commands rdmsr (read msr) to read the actual clock modulation and wrmsr (write msr) to write and change the clock modulation.

We decided to test this code inside the Thread Idle method, like this:

Write and Read

This code was made before the clock (Hertz new_clock) method be done on CPU code.

```
unsigned long long modulation = 0b10011;
    unsigned long long clock_m_read;
    unsigned long msr = 0x19a;
    bool changed = false; //used only to avoid consecutive write operations on MSR
    while(_thread_count > Machine::n_cpus()) { // someone else besides idles
        if(Traits<Thread>::trace_idle)
            db<Thread>(TRC) << "Thread::idle(CPU=" << Machine::cpu_id() << ",this=" <<
running() << ")" << endl;
        if (Machine::cpu_id() == 0) {
                db<Thread>(WRN) << "THERMAL_STATUS: "<< Thermal::temperature()<<endl;
                if (!changed) {
                                asm volatile ("rdmsr": "=A"(clock_m_read): "c"(msr));
                                db<Thread>(WRN) << "FREQUENCY -> "<< clock_m_read << " <-
"<< CPU::clock() <<endl;
                                asm volatile ("wrmsr" : : "c"(msr), "A"(modulation));
                                asm volatile ("rdmsr": "=A"(clock_m_read): "c"(msr));
                                db<Thread>(WRN) << "FREQUENCY AFTER ALTER -> "<<
clock_m_read << "<- "<< CPU::clock() <<endl;
                                changed = true;
                        }
        }
        CPU::int_enable();
```

```
        CPU::halt();
        if(_scheduler.schedulables() > 0) // A thread might have been woken up by another
CPU
            yield();
    }
```

Result

First Alteration and Read (Before->After)
After '->' the first number is the IA32_CLOCK_MODULATION value and after '<-' the number indicates the value of CPU clock() method return.

```
<0>: FREQUENCY -> 00 <- 33923534430 :<0>
<0>: FREQUENCY AFTER ALTER -> 19 <- 3392353440 :<0>
```

As we can see on the result, the CPU clock() value is unchanged even with the modulation change. So, to test if the change occurs correctly, we used a code placed on the appĺication main that takes the value of EPOS RTC date() in the beggining of the execution and, just before the return, takes another read of RTC date() and subtract the previous (beggining) read. Part of this code is placed below.

```
int main()
{
    unsigned long long date_1 = RTC::date();
    ...
    cout<<"RTC"<<RTC::date() - date_1<<endl;
    return 0;
}
```

With a code that only reads the clock modulation or do not do anything on Thread idle() the printed value is 1, as it's placed on the following results

```
The end!
RTC1
<0>: The last thread has exited! :<0>
<0>: Rebooting the machine ... :<0>
```

But, if the code that writes the value to the modulation is used the tests printed values 4 and 5 on the 2 tests executed:

```
RTC5
<0>: The last thread has exited! :<0>
<0>: Rebooting the machine ... :<0>
```

```
RTC4
<0>: The last thread has exited! :<0>
<0>: Rebooting the machine ... :<0>
```

Code on CPU: :clock(Hertz new_clock);

Here we have made a code that given a desired clock, transform into one of the classes available in

IA32_CLOCK_MODULATION MSR with the bit 0 not reserved and write the MSR with the calculated modulation.

```
static void clock(Hertz new_clock) {
        Reg32 clockm_addr = 0x19a;
        float fator;
                if (new_clock <= _cpu_clock * 0.0625) {
                        fator = 0.0625;
                } else {
                        float aux = 1./_cpu_clock;
                        fator = new_clock*aux;
                }
                Reg64 modulation;
                if (fator > 0.9375) {
                        modulation = 0b01000;
                } else {
                        fator *= (100/6.25);
                        Reg64 ifator = (Reg64) (fator);
                        if ((fator - ifator) > 0) {
                                ifator += 1;
                        }
                        modulation = 0b10000 | ifator;
                }
                wrmsr(clockm_addr, modulation);
    }
```

Heuristics

Thermal Control

To control temperature aspects of each core we used a code placed on dispatch method (into thread.cc). The actuation limits of the "controller" were setted to be short and on a low temperature because the test used do not stress to much the architecture (this is only a first test version):

Code used:

```
void Thread::dispatch(Thread  prev, Thread  next, bool charge)
{
    if(charge) {
        if(Criterion::timed)
            _timer->reset();

        //ADVFS
        if(Criterion::energy_aware) {
            _cpu_temperature[Machine::cpu_id()] = Thermal::temperature();

            //set max temp
            if(_cpu_temperature[Machine::cpu_id()] >= 45) {
                unsigned long new_hz = 896146304; //25 - 30% of the normal clock
                CPU::clock(new_hz);
            } else if (_cpu_temperature[Machine::cpu_id()] <= 40) {
```

```
                CPU::clock(CPU::clock());
            }
        }
    }
...
}
```

Last cores's temperature read without Thermal "controller":

```
<2>: CPU2 Temperature = 59 :<2>
<3>: CPU3 Temperature = 58 :<3>
<1>: CPU1 Temperature = 56 :<1>
<0>: CPU0 Temperature = 56 :<0>
<6>: CPU6 Temperature = 56 :<6>
<7>: CPU7 Temperature = 56 :<7>
<5>: CPU5 Temperature = 55 :<5>
<4>: CPU4 Temperature = 55 :<4>
```

Last cores's temperature read with Thermal "controller":

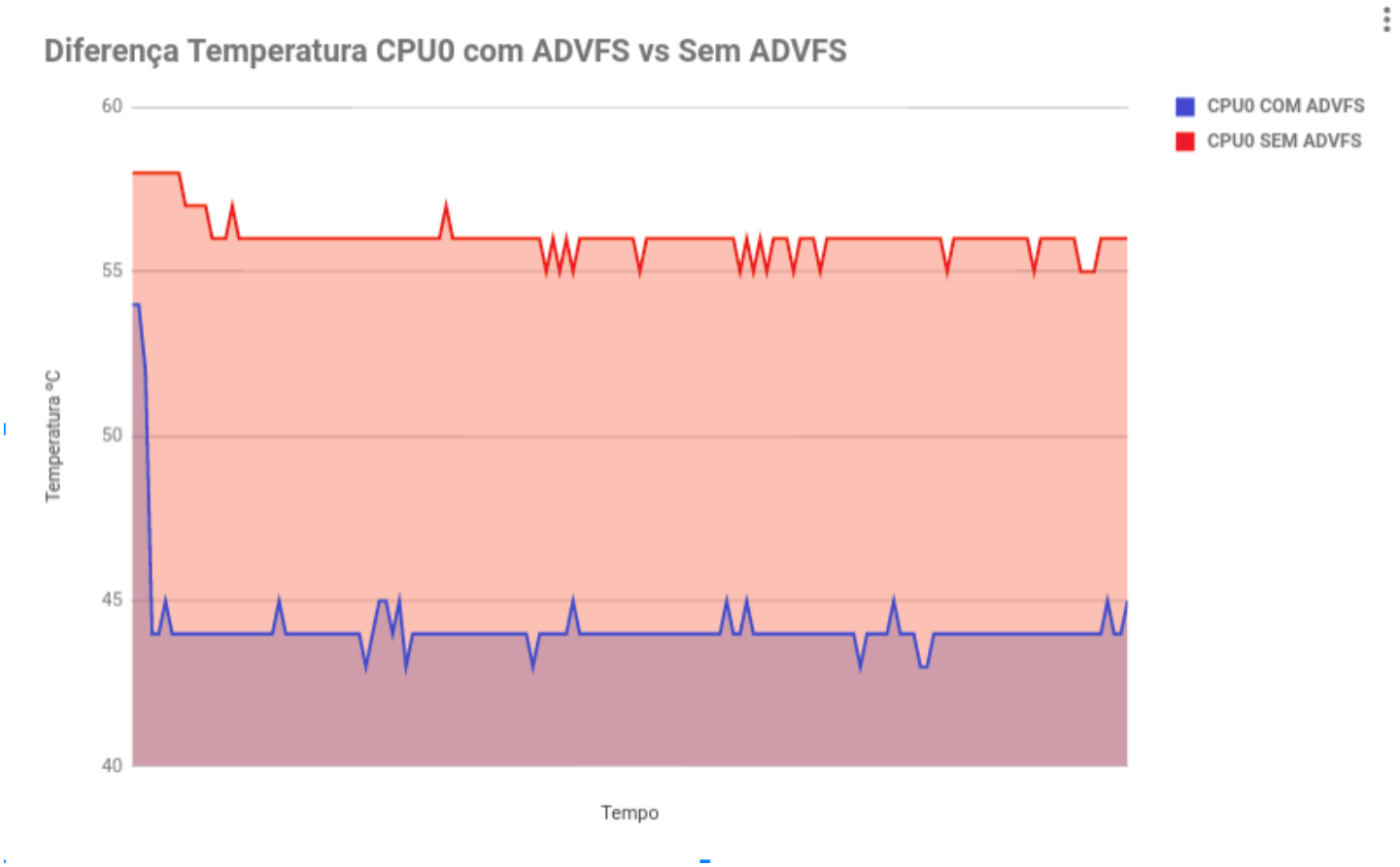
```
<6>: CPU6 Temperature = 43 :<6>
<7>: CPU7 Temperature = 43 :<7>
<4>: CPU4 Temperature = 46 :<4>
<5>: CPU5 Temperature = 45 :<5>
<1>: CPU1 Temperature = 43 :<1>
<2>: CPU2 Temperature = 46 :<2>
<3>: CPU3 Temperature = 44 :<3>
<0>: CPU0 Temperature = 43 :<0>
```

Temperature Charts

We now show the temperature comparison on 2 of the 8 (CPU 0 and 3) Cores of the PC:

Temperature on CPU0

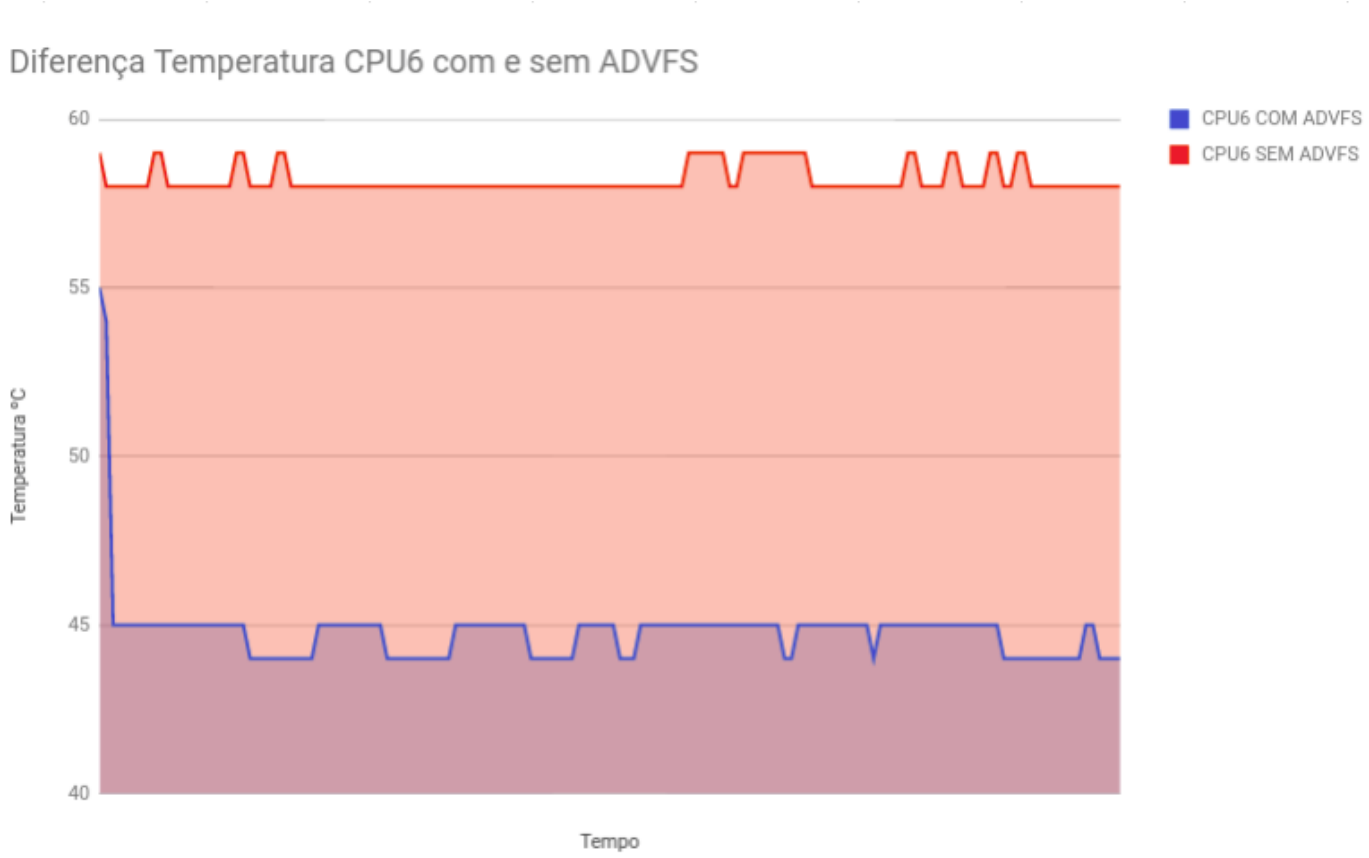
English legend: Temperature Difference of CPU 0 with and without ADVFS.
Blue color: with ADVFS
Red color: without ADVFS
x axis: Time
y axis: Temperature °C



The chart legend is in portuguese because of a presentation to our classmates.

Temperature on CPU6

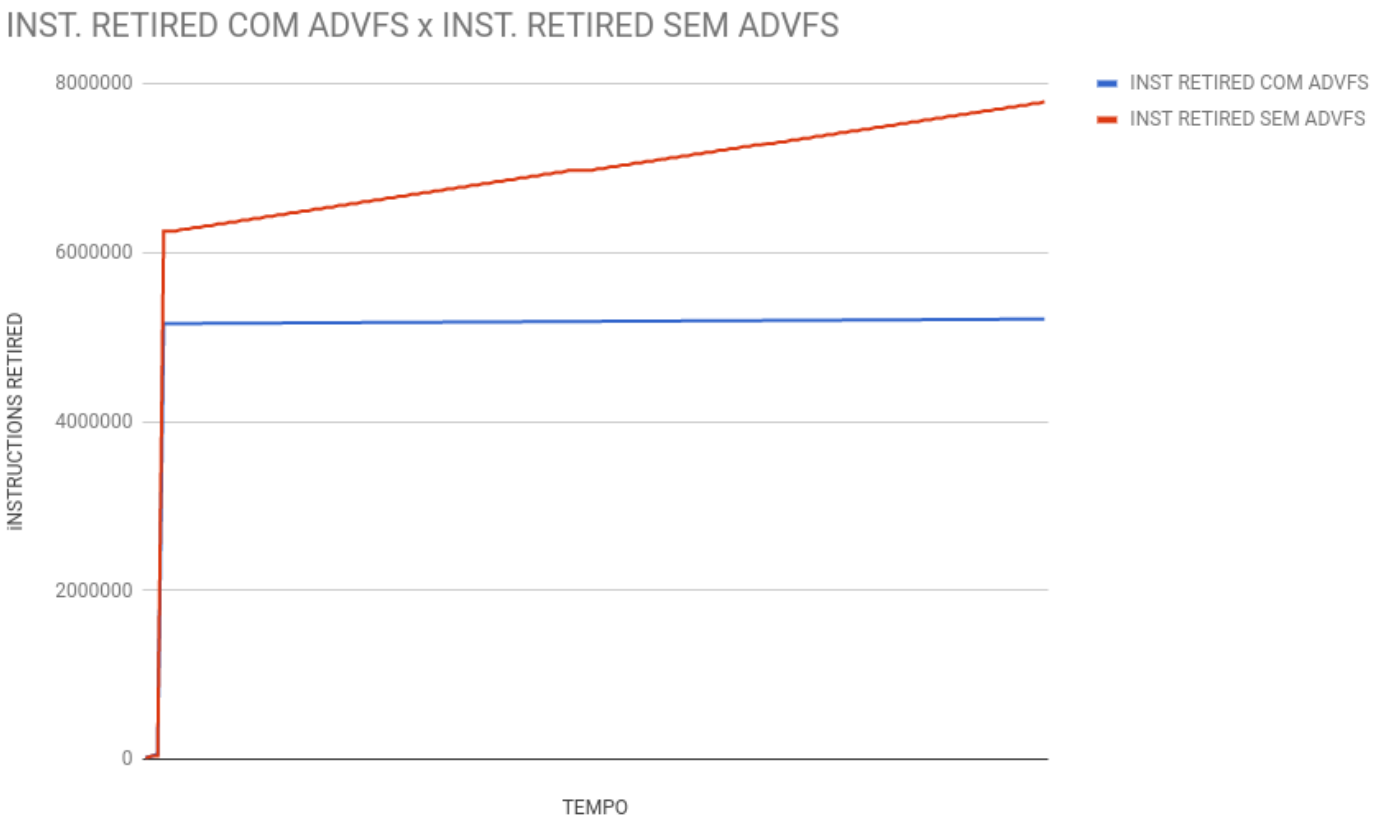
English legend: Temperature Difference of CPU 6 with and without ADVFS.
Blue color: with ADVFS
Red color: without ADVFS
x axis: Time
y axis: Temperature °C



The chart legend is in portuguese because of a presentation to our classmates.

Instruction Retired Chart

Chart legend: INST RETIRED with ADVFS and INST RETIRED without ADVFS
Blue color: with ADVFS
Red color: without ADVFS
x axis: TIME
y axis: INSTRUCTION_RETIRED



Bibliography

1. WILLHALM, T. (Intel), DEMENTIEV, R. (Intel); FAY, P. (Intel), **Intel® Performance Counter Monitor - A better way to measure CPU utilization**, Updated January 5, 2017. Available on: https://software.intel.com/en-us/articles/intel-performance-counter-monitor#example . Accessed on: 12/09/2017,
2. **Intel® Turbo Boost Technology 2.0**. Available on: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html . Accessed on: 12/09/2017.
3. **Intel® 64 and IA-32 Architectures Software Developer's Manual**, Available on: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf . Accessed on: 12/09/2017.
4. **Eclipse IDE C/C++**, Available on: http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/heliossr2 . Accessed on: 12/09/2017.
5. SULEIMAN, D., IBRAHIM, M., HAMARASH, I., **DYNAMIC VOLTAGE FREQUENCY SCALING (DVFS) FOR MICROPROCESSORS POWER AND ENERGY REDUCTION**, Available on: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=E6F55A8CE6B176124DC60E15141B65B5?doi=10.1.1.111.1451&rep=rep1&type=pdf . Accessed on: 20/09/2017.
6. DEHMELT, F., **Adaptive (Dynamic) Voltage (Frequency) Scaling—Motivation and Implementation**; Application Report SLVA646; Texas Instruments, 2014. Available on: http://www.ti.com/lit/an/slva646/slva646.pdf . Accessed on: 21/09/2017.
7. SNOWDON, D., RUOCCO, S., HEISER, G., **Power Management and Dynamic Voltage Scaling: Myths and Facts**. National ICT Australia and School of Computer Science and Engineering University of NSW. Sydney, 2005. Available on: http://www.snowdon.id.au/Publications_files/snowdon05power2.pdf . Accessed on: 22/09/2017.
8. BLOCK, A., WILLIAM KELLEY, W., **Adaptive Clustered EDF in LITMUS**; Austin College. Sherman, Texas; BAE Systems. Ft. Worth, Texas. Available on:

https://people.mpi-sws.org/~bbb/events/ospert15/pdf/ospert15-talk-s3.pdf . Accessed on: 24/09/2017.

9. **Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2**, Available on: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf . Accessed on: 29/09/2017.
10. DONYANAVARD, B.; MÜCK, T.; SARMA, S.; DUTT, N., **SPARTA: Runtime task allocation for energy efficient heterogeneous manycores**, Department of Computer Science, University of California, Irvine, USA, IEEE, 2016. Available on: http://ieeexplore.ieee.org/document/7750975 . Accessed on: 29/09/2017
11. MÜCK, T.; SARMA, S.; Dutt, N.; **Run-DMC: runtime dynamic heterogeneous multicore performance and power estimation for energy efficiency**, Amsterdam, The Netherlands, IEEE, 2015. Available on: https://dl.acm.org/citation.cfm?id=2830859 . Accessed on: 29/09/2017.
12. **How to get the values of on die digital thermal sensors for Atom D510?**. Available on: https://communities.intel.com/thread/43687 . Accessed on: 04/10/2017
13. BERKTOLD, M., TIAN, T., **CPU Temperature Monitor: Intel DTS and PECI**. Available on: https://www.intel.com/content/www/us/en/embedded/testing-and-validation/cpu-monitoring-dts-peci-paper.html . Accessed on: 02/10/2017.
14. SALZMAN, P., BURIAN, M., POMERANTZ, O., **The Linux Kernel Module Programming Guide**, 2007-05-18 ver 2.6.4. Available on: http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html#AEN121 . Accessed on: 09/10/2017.