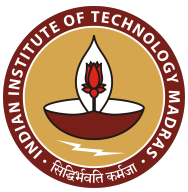

Indian Institute of Technology Madras

Department of Computer Science & Engineering

CS3500 Operating Systems

Course Project Report

A Study of Linux's EAS and Possible Algorithmic Improvements

Arnhav Abhijit Datar - CS18B003

C Shriram - CS18B007

December 7, 2020

Contents

1 Introduction **2**

2 Literature Review **2**

3 Understanding `kernel/sched/fair.c` **3**

4 Basic Functions **3**

4.1 Frequency . . . 3

4.1.1 Algorithm . . . 3

4.1.2 Code . . . 3

4.2 Temperature . . . 4

4.2.1 Algorithm . . . 4

4.2.2 Code . . . 4

5 Heuristics and Code **5**

5.1 Tie Breaker Using Frequency . . . 5

5.1.1 Heuristic . . . 5

5.1.2 Code . . . 5

5.2 Frequency Adjustments . . . 6

5.2.1 Heuristic . . . 6

5.2.2 Code . . . 6

5.2.3 Helper Functions . . . 6

5.3 Tie Breaker Using Temperature . . . 7

5.3.1 Heuristic . . . 7

5.3.2 Code . . . 7

6 Author Contributions **8**

Introduction

Energy Aware Scheduling is a scheduling paradigm that gives the process scheduler the power to rearrange processes on different CPUs based on energy consumed by the CPUs and processes. The Linux EAS is implemented to accurately select an energy efficient CPU for each process, effectively decreasing the energy consumption of a computer system in the process.

To put in simpler words, the EAS paradigm is an energy efficient replacement to the way the default CFS scheduler of Linux assigns processors to each process. More on how these processors are assigned and what improvements can be done here to improve the performance of the computer are analysed in this project.

It must be noted that EAS works only on computer systems that have asymmetrical multi-processor systems, for example, ARM's big.LITTLE.

Literature Review

1. We have familiarized ourselves with the given paper[1] on EAS. The block diagram of the process in the paper can be seen in Figure 1.

2. We read the chapters related to scheduling in Professional Linux Kernel Architecture Wolfgang Maurer.

3. We also went through the Linux documentation for the Energy Aware Scheduler[2] and the chromium OS project's EAS resources[3].

4. We installed the Linux kernel 5.6.9 on an Intel-Ubuntu 18.04 computer to write and test code for the scheduler.

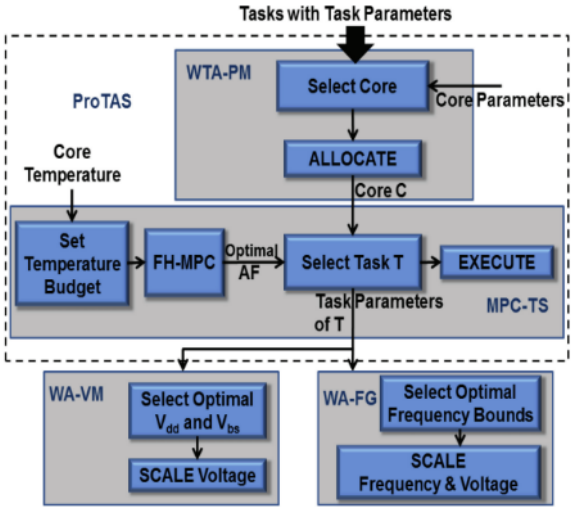

Figure 1: Block diagram: ProWATCh

Understanding kernel/sched/fair.c

We worked on the Linux 5.6.9 kernel. We first developed a function call graph to understand the default scheduler of Linux. We discovered that the default EAS of Linux is mainly linked to the function `find_energy_efficient_cpu()` in `kernel/sched/fair.c`. The rest of the report deals with a few upgrades that can be done to this function to improve the performance of Linux EAS.

Basic Functions

Frequency

Algorithm

The `getfreq()` is simple and directly uses a function from `kernel/sched/sched.h`. The `setfreq()` function follows the following procedure:

1. First, finds the CPU frequency policy.
2. Adjusts the frequency to be within the bounds given by the frequency policy for the processor.
3. Changes the frequency of the concerned processor to the closest available frequency using the `CPUFREQ_RELATION_C` flag which assigns the closest possible frequency to the CPU.

Code

Given below are our implementations for the functions to set and get CPU frequency.

Listing 1: Set and Get frequency Methods

```
1  int getfreq(int cpu)
2  {
3      int frq=(int)cpufreq_get(cpu);
4      return frq;
5  }
6  void setfreq(int cpu, unsigned int modfreq)
7  {
8      //First we get the CPU Frequency policy
9      struct cpufreq_policy *policy=NULL;
10     cpufreq_get_policy(policy, cpu);
11
12     //If we get a reasonable frequency policy.
13     if(policy!=NULL)
14     {
15         //Ensuring that it is within limits
16         modfreq=min(policy->max, modfreq);
17         modfreq=max(policy->min, modfreq);
```

```
18
19          //Changing the frequency
20          __cpufreq_driver_target(policy, modfreq, CPUFREQ_RELATION_C);
21          //CPUFREQ_RELATION_C gets the closest frequency to the target ↩
                frequency(modfreq)
22      }
23  }
```

Temperature

Algorithm

We require functions to access CPU temperature, which is done here using the Intel MSR registers. We use the `rdmsrl_safe_on_cpu()` function defined in `arch/x86/include/asm/msr.h` along with the `MSR_IA32_THERM_STATUS` flag, which returns a value who's 31st bit is the validity bit and the 16th through 22nd bits give the temperature of the CPU in the range from 0 to 255. MSR Registers Documentation[4] was used for the same. The diagram below indicating the bits in the register

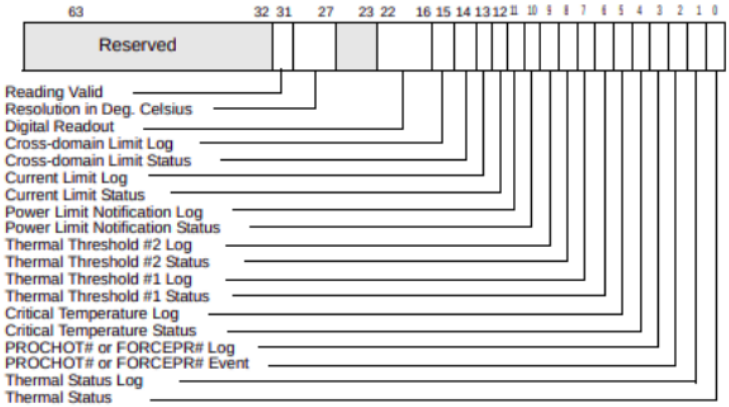

Figure 2: IA32_THERM_STATUS register

Code

Listing 2: Helper Functions: gettemp

```
1   int gettemp(int cpu)
2   {
3       //Get the temperature
4       uint64_t temp=0, vfl, msk;
5       rdmsrl_safe_on_cpu(cpu,
6       MSR_IA32_THERM_STATUS, &temp);
7
8       // A mask to check validity
9       vfl=1<<31;
```

```
    //A mask to get only the temperature
    msk=1111111;

    //Check for validity
    if(temp&vfl){
        //Take bits 22:16
        temp=temp>>16;
        temp= temp&msk;
        // return the temperature
        return temp;
    }
    else
        return -1;
}
```

Heuristics and Code

There are a few blocks in which optimisations can be done on the already existing `find_energy_efficient_cpu()` function to improve Linux's EAS. A few of them, as implemented below, are listed:

Tie Breaker Using Frequency

Heuristic

In the default EAS implementation of Linux, there is no tie breaker in choosing efficient CPUs. We seek to improve this by giving the task to the highest frequency CPU among the CPUs with the largest spare capacity, if the contender CPU has the same performance domain. This update has the capability to improve the speed of the system by assigning CPUs more efficiently to processes.

Code

In `find_energy_efficient_cpu`, while we iterate over all CPUs while comparing the spare_caps, we set higher frequency as a tie-breaking policy

Listing 3: Tie-Breaking Heuristic 1

```
    // best_freq and freq are initially initialised to -1
    freq = getfreq(cpu);
    if ((spare_cap > max_spare_cap)
        || (spare_cap == max_spare_cap && freq > best_freq)) {
        //Tie breaking Condition
        max_spare_cap = spare_cap;
        max_spare_cap_cpu = cpu;
        best_freq = freq;
```

```
9           //Updating best_freq
10      }
```

Frequency Adjustments

Heuristic

We notice that when we move a particular task from one CPU to another, this reduces the load of the original CPU and increases the load on the newer efficient CPU to which the process is now moved. This may lead to a higher power consumption overall. To remedy this while preserving efficiency, we increment the frequency of the old CPU that just lost the process and decrement the frequency of the new, more efficient CPU that gained a process.

Code

Given below is the code for the heuristic in `kernel/sched/fair.c`. Just when we are about to return the best CPU to `select_task_rq_fair` we decrement frequency of *best_energy_cpu* and increment frequency of *prev_cpu*.

Listing 4: Frequency adjusting heuristic

```
1       // if prev_cpu cannot be used
2       if (prev_delta == ULONG_MAX){
3           decfreq(best_energy_cpu);
4           incfreq(prev_cpu);
5           return best_energy_cpu;
6       }
7       // or if it saves at least 6% of the energy used by prev_cpu.
8       if ((prev_delta - best_delta)>((prev_delta + base_energy) >> 4)){
9           decfreq(best_energy_cpu);
10          incfreq(prev_cpu);
11          return best_energy_cpu;
12      }
```

Helper Functions

For the heuristic above, we need functions to increment and decrement the frequency of a CPU which are implemented below:

Listing 5: Helper Functions: Incfreq and Decfreq

```
1   void incfreq(int cpu){
2       int frq;
3       frq=(int)getfreq(cpu);
4       frq+=frq>>4;
```

```
    // We increment the frequency by 6.25% and set it below
    setfreq(cpu, frq);
}
void decfreq(int cpu){
    int frq;
    frq=(int)getfreq(cpu);
    frq-=frq>>4;
    // We decrement the frequency by 6.25% and set it below
    setfreq(cpu, frq);
}
```

The code for these functions is simple - we get the frequency of the CPU in notice using the basic functions defined earlier. We then increment/decrement this frequency by a factor of 6.25% using right shift operators. We then set the corresponding frequency back to the CPU.

Tie Breaker Using Temperature

Heuristic

Similar to heuristic 1, this heuristic improves on the tie breaking condition. The process is assigned to the processor with the lowest temperature among the processors that have the maximum capacity.
By choosing the CPU with a lesser temperature, this ensures that makes sure there is no slowing down of the processor due to heating. This is also safer for the CPUs since it reduces chances overheating and spoilage due to high temperature radiation.

Code

In `find_energy_efficient_cpu`, while we iterate over all CPUs while comparing the spare_caps, we set lower temperature as a tie-breaking policy

Listing 6: Tie-Breaking Heuristic 2

```
    // best_temp and temp are initially initialised to 256
    temp = gettemp(cpu);
    if ((spare_cap > max_spare_cap)
        || (spare_cap == max_spare_cap && temp < best_temp)) {
        //Tie breaking Condition
        max_spare_cap = spare_cap;
        max_spare_cap_cpu = cpu;
        best_temp = temp;
        //Updating best_temp
    }
```

Author Contributions

While the coding was primarily done by Arnhav Datar since there was only one working Linux kernel, all functions and heuristics were discussed and done collaboratively between Arnhav Datar and Shriram Chandran.

References

[1] Milan Patnaik, Chidhambaranathan R, Chirag Garg, Arnab Roy, V. R. Devanathan, Shankar Balachandran, and V. Kamakoti. Prowatch: A proactive cross-layer workload-aware temperature management framework for low-power chip multi-processors. *J. Emerg. Technol. Comput. Syst.*, 12(3), September 2015.

[2] Energy aware scheduling. https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html.

[3] Energy aware scheduling- chromium project. https://www.chromium.org/chromium-os/obsolete/energy-aware-scheduling.

[4] Intel documentation for msr. https://software.intel.com/content/dam/develop/external/us/en/documents-tps/335592-sdm-vol-4.pdf.