Technion – Israel Institute of Technology, EE Faculty Spring 2015

Improving Linux Load Balancing

Written by Boaz Haim
Supervised by Noam Shalev

Abstract

Linux is one of the biggest open source operating systems (O.S) in the world. The Kernel is the brain of the O.S. In this project we studied the Load Balancing operation in order to understand how the Kernel behaves when one of the CPU's shuts down while the system is loaded with tasks. We have seen that the Kernel uses a simple abstraction for load balancing and in it there is no consideration for offline CPU's. A new system call was added, which monitors the CPU's and gives a sample of each CPU run queue size. Then we changed the load balancing helper functions, to make them calculate the load over each CPU in the correct way, thus we take out of the equation the offline CPU's and make sure the Kernel will perform a better load balancing when one of the CPU's goes offline.

Contents

List of figures

Introduction

Linux Kernel is the brain of Linux, it controls all the needed operation to make sure that Linux is working correct and fast. Most if not all of the computers today have more than 1 CPU and the kernel needs to know how to use them in a smart way. In this project we will show that when the computer has many tasks and one of the CPU's goes offline, the load balancing operation is not working as we would expect. Hardware failure is not something new in the computer world, thus we want the Kernel to be able to perform the load balancing in a way that would cause a proper load balancing.

We started the project with 3 interesting missions:

1. Create a tool and prove with it that the problem truly exists.
2. Study, analyze and fix the Load Balancing operation.
3. Use the tool from Chapter No. 1 to validate that the solution works.

This book describes the work that was done on the project. **Chapter 1** talks about the problem in the Kernel, shows the creation of the tool that proves the problem existents. **Chapter 2** shows the learning process and gives a detailed explanation of data structure and function that are used in the load balancing operation. **Chapter 3** provides details about the changes that were done, covering all the data structures and the functions. In **Chapter 4**, the results are shown and discussed after doing all the changes in the Kernel. **Chapter 5** summarizes the work and the project.

Chapter 1: Motivation & prove of problem existents

Basic concepts

Linux is an open source operating system. The Kernel is the brain of the system, one of its roles is to make sure that all the tasks that are in the system are balanced among all the CPU's. When a task is created, it is automatically added to CPU from which it's father is running on. If the task stays on that CPU it can cause an unwanted situation where one CPU has many tasks waiting for CPU time, when other CPU's has no work at all.

Project Motivation

Current implementation of the Load Balance operation does not take under consideration offline CPU's. This can cause a case where the system stays unbalanced at all. We will show that if the system is balanced and one of the CPU's stops working, the CPU's workload is unbalanced and some CPU's will have more work than others.

Building a tool for monitoring the CPU's work

In this task we needed to prove that the problem. The first step was to add a system call that does a simple task: every predefined period, go over all the CPU's in the system and print to the log the amount of tasks that they have waiting in the run queue. For implementing the system call we used Kernel timers.

```
for_each_present_cpu(cpu)
{
    i++;
    curRq=cpu_rq(cpu);
    curRunablle=curRq->nr_running;
    printk("boaz in get_cpu_workload-round %d:cpu num is %d and number of proc is %d\n",roundNumber,i,curRunablle);
}
if (roundNumber < roundsToPerform)
{
    setup_timer(&my_timer,get_cpu_workload,0);
    ret = mod_timer(&my_timer,jiffies + msecs_to_jiffies(SLEEP_TIME));
}
printk("boaz in get_cpu_workload- roundNumber is %d\n",roundNumber);
roundNumber++;
```

Figure 1-The loop that goes over all the CPU's

As can be seen in Figure1, we are using a Kernel macro – *for_each_present_cpu* that goes over all the CPU's in the system. For each one we are printing the state. After that we check if the user wants to check the system state more times, if he did we are setting again the timer that would call the same function. During this project the printing time was set to 3 millisecond.

Once the system call was ready, we had to create a loaded system. We did this by creating a small C program that performed a fork operation for a few times and then, each task goes over a loop for 1500 times. The reason that the tasks performed the loop and no other operations like Sleep or Wait is that

we needed the tasks to be ready for execution in order to see them on the run queue, if a task is sleeping or waiting we won't then since they are not ready for execution.

At this we point we had the Kernel system call and an executable that loads the system. We created a Bash script that controls everything. It starts by calling the system load and starting the system watch (the system call), then it shuts down one of the CPU's, waits and starts the CPU again.

```
#!/bin/bash

#start loading with process
./bashScriptForBusyWait &

#start monitoring the cpus
./testSyscall -1 366 &

#sleep for 1 sec
./myOwnSleep 1

#shut down cpu1
echo "going to shut down cpu1"
echo 0 > /sys/devices/system/cpu/cpu1/online

#wait for another 4 sec
./myOwnSleep 4

#wake up cpu1
echo "going to wake up cpu1"
echo 1 > /sys/devices/system/cpu/cpu1/online
```

Figure 2-Bash script for controlling the test

Testing the Kernel & Results analysis

To analyze the results, we needed to look at the Kernel log, which can be seen by calling DMESG from a terminal and then filtering the messages that came out from our functions. All the tests were done on an Asus computer, i5 processor and 4GB RAM.

At first, we can see that the tool does work and it gives a snapshot of the run queues at each sampling time. When processing all the results we get the following graph:

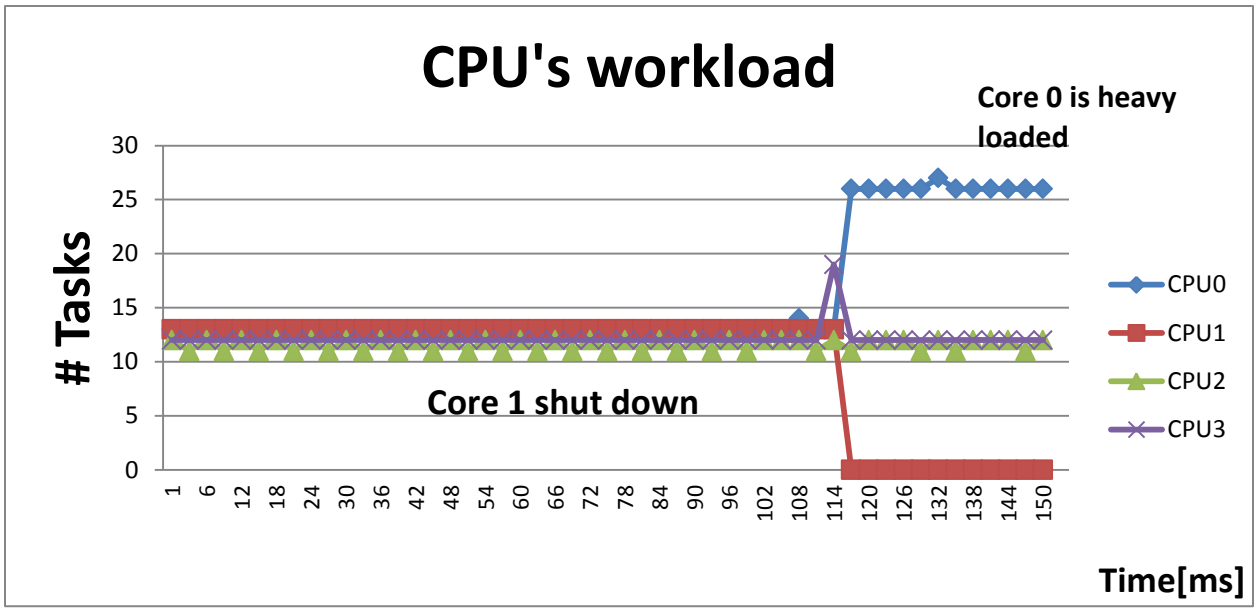


Figure 3-System state when CPU is down

As we can see in the graph, the system is in stable mode until we shut down CPU1. Once the CPU is down, all of the jobs are taken to CPU0, and they stay on it. Thus causes CPU0 to have much more work than the rest of the system.

We can see this issue also in the log prints:

```
cpu num is 1 and number of proc is 15
cpu num is 2 and number of proc is 18
cpu num is 3 and number of proc is 17
cpu num is 4 and number of proc is 19


cpu num is 1 and number of proc is 32
cpu num is 2 and number of proc is 0
cpu num is 3 and number of proc is 19
cpu num is 4 and number of proc is 22
```

Figure 4-System state when shutting down a CPU

In Figure 4 we see the system state at the moment that CPU shuts down. Before the shut down the system is in a balanced state. At the moment the CPU is down, all of the work moved to CPU1 and as we can see in the next figure, this state stays the same for a long time.

```
:cpu num is 1 and number of proc is 33
:cpu num is 2 and number of proc is 0
:cpu num is 3 and number of proc is 18
:cpu num is 4 and number of proc is 21
```

Figure 5-System state after a long time

Chapter Conclusions

As seen in this chapter we have proved that the problem truly exists and can cause many performance issues on Linux once we have a system that one of its CPU's goes offline. Since the CPU's are in the Kernel side there is a need to add a new system call into the system that monitors the CPU's run queue.

Chapter 2: Study & Research

Introduction

In this chapter we will talk about the Load Balancing function and data structures that are being used in the balancing operation. During this chapter we will get a clear and deep understanding of how it works, who calls the load balance and all the chain reactions that happens, starting with the hardware interrupt and ending in the tasks movement from CPU to CPU.

Data structures

Overview

The Load balancing operation uses several data structures. Some of them might look unneeded but the reason to have them is to collect all the data in small groups of data structures instead in one big structure. By doing this serration it helps to hold only the relevant information at each time.

Sched domain

Sched_domain is the most common data structures being used in the load balancing operation. Each domain represents a number of CPU's, the Kernel holds a tree of Sched domains, when each Sched domain has under it more Sched domains which represent less CPU's, as can be seen in the next figure:

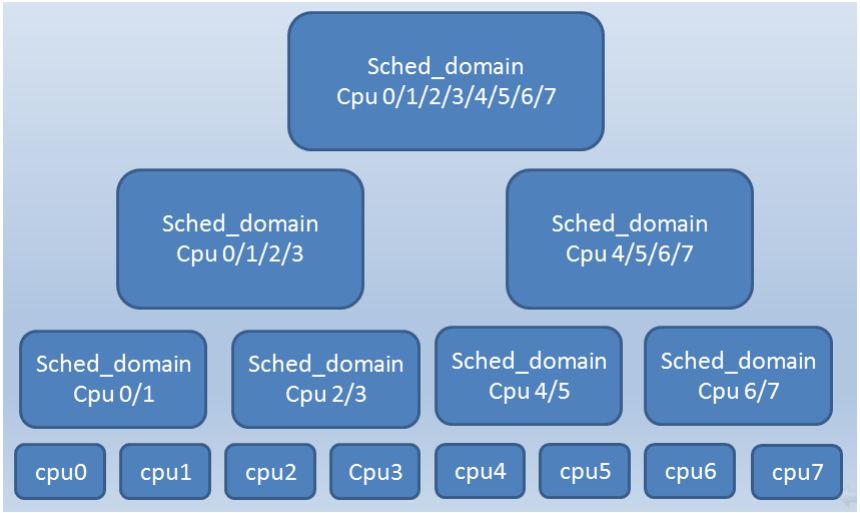


Figure 6-sched domain structure

Figure 6 gives an example of the way the Kernel uses the Sched domain data structure. When looking only at the CPU's, each one of them stands on its own. But every group of them (in Figure6 it's in groups of 2) is represented by one Sched domain, which has other siblings. At the end the Kernel looks at one Sched domain which represents all CPU's.

Sched group

The Sched group is being hold inside the Sched domain, and it contains the actual representation of the CPU's and each Sched domain can contain several Scehd groups, depending on the system type. Also contains other important parameters like group power, which represents the amount of tasks that the group can take.

Lb env

Load balance environment. Helps to represents a group of CPU's we consider for the load balance. As seen in the following figure.

```
struct lb_env {
    struct sched_domain *sd;

    struct rq       *src_rq;
    int          src_cpu;

    int          dst_cpu;
    struct rq       *dst_rq;
```

Figure 7-A few fields from the lb env structure

Sd lb stats

Sched domain load balance statistics. Stores the statistics of a Sched domain during the load balancing operation. Contains several important statics, for example it contains the average load of the domain.

Sg lb stats

Sched group load balance statistics. Also stores statistics as the Sched domain data structure, statistics that are relevant for the group balancing operation.

How does load balance starts

There are two ways to trigger the Load Balance:

1. When the current run queue is empty.
2. Pre-defined timer: every 1 millisecond if the system is in idle state or every 200 millisecond otherwise.

For the purpose of our project we have only the second situation, in which the system is busy and thus the Load Balance is triggered every 200 milliseconds by a timer.

When the timer wakes up, it calls a function named Run Rebalance Domains, which goes over each domain in the system and tries to balance it with the help of load balance.

Load balance functions call

Should we balance

The first function that is called from the load balance method. The function role is to decide if there is a need to perform Load Balancing - this is done by going over every CPU in the group and check if there is a case of CPU that is working more than the any other CPU in the group. If there is a case like this, we do need to perform load balancing and the function returns 1, otherwise it returns 0.

Find busiest group

Goes over all the groups in the domain and searches for the group that has more job to do than others. The check is being done by comparing the group's average work. If it does find a candidate group, it will also call a side function that calculates the amount of job that needs to be moved from this group, in order to have a balanced system. There is also a small check to see if the amount of work to move is very small, if it does we call a little macro that moves the jobs (this prevents from doing many tasks and only moving one or two tasks).

Task h load

The function calculates the relevant load of the task. Since the kernel uses Complete Fair Scheduling, it uses this function to get the amount of work that the tested task uses in the system.

Move tasks

Once the load balancer knows the destination and source CPU's, it calls to move tasks with them. Move tasks uses the amount of job that needs to be moved (that was calculated in find busiest group) and tries to move that amount of jobs from the source CPU to the destination CPU. Before moving a task the function performs several checks, one of them uses the result from task h load to see if the task load/2 is bigger than the amount of job that needs to be moved .If the size is big the current task won't be moved, thus not moving a big task.

Overall flow

The following scheme shows the main flow that causes the Load balancing operation.

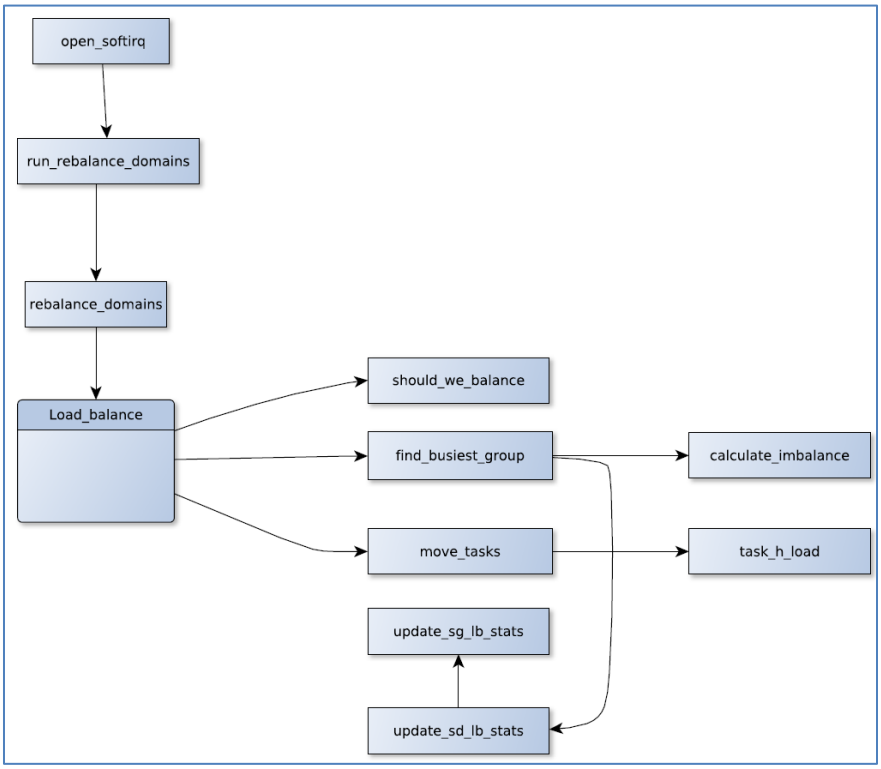


Figure 8 - Load Balance flow

After reviewing the process, we can understand where exactly the problem is.

As demonstrated in the previous chapters, the Kernel uses the average work to decide the amount of job to take during the operation. The problem is that this calculation does not take under consideration offline CPU's.

Chapter 3: Suggested solution, Development & Implementation

Introduction

This chapter presents the suggested solution. It also describes the relevant changes that were done to the kernel and their implementation.

Suggested solution

The Kernel uses an abstraction to perform the Load Balance operation. If we take all the places where the average is calculated, or taken under consideration. We replace the calculation with a new one that considers offline CPU's.

The solution will also be a "simple" one: we won't take all the special cases that exists in the Kernel (like the type of the hardware and more).

The behavior we would like to see:

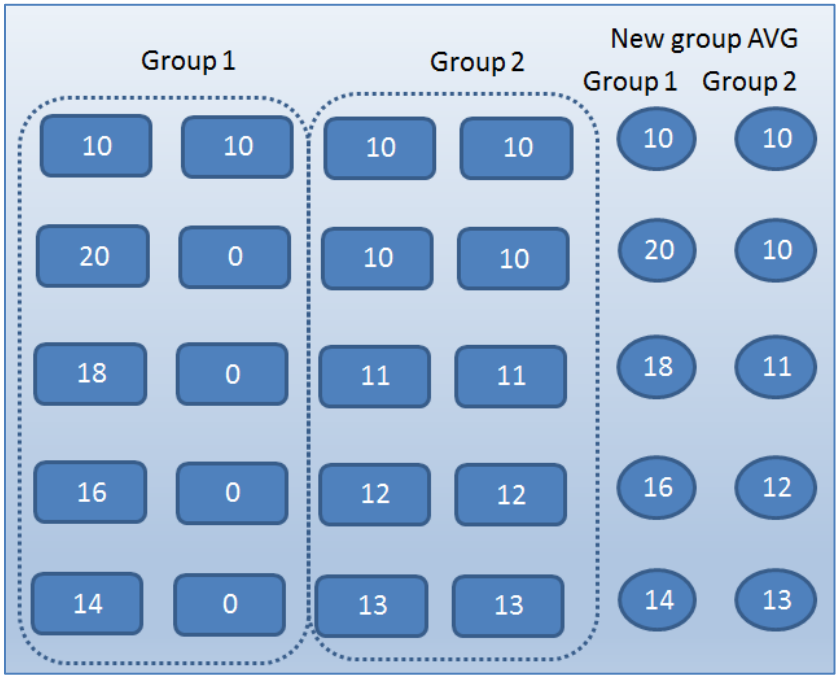


Figure 9- desired average calculation

If we look at a system that has 4 CPU's and each one has 10 tasks waiting, the moment one CPU is shutdown, the average starts to be imbalanced and task starts to move to the other group, until the system is almost balanced.

Implementation overview

First Implementation attempt

When looking again at the data structures and the functions, you can see that two of the data structures have the average load in them: Sd_lb_stats & Sg_lb_stats. Each average presents the average work in each Sched domain or Sched group. During this test we believed that if we change the initiation & update function of each data structure, it would cause a chain reaction and will fix the problem.

After updating the data structures, the initiation and the update functions to calculate only the average work with the online CPU's in each group, the system had not changed and everything kept working the same. All the outputs from the tool gave the same result as if nothing was changed.

At this point, we understood that this bottom up approach won't work and a new way of thinking needs to be done.

Final implementation

We started this approach with a different way of thinking and decided that the solution would need to be done from up to down. We needed to go over the load balancing operation and check each function call and data structure use, and to understand if the average work has any kind of influence on it. When we did found something that had influence, there was a need to do the calculation again taking under consideration the offline CPU's. In most of the code, there was only a need to recalculate the average work since there are many points in the code that calculate the average at the point that they were called instead of using the average that is in the statistics data structures (and this is why the first attempt did not succeeded).

Beside the average work calculations, we had also to change the way that the system calculates the imbalance. The imbalance represents the amount of work that should be moved from on CPU to the other in order to achieve balanced system. The current calculation took under consideration many things like the type of the CPU, the amount of work that can be added to it and more. We had to simplify the calculation and take out of the question some info that was added to it in the original calculation. A few more minor changes were done and would be discussed in the next section.

Detailed Implementation

Most of the changes were done in the functions. We will go over the main functions and their changes.

Should we balance

We had to change to way we check the average, this is why we added two counters in the loop that goes over each CPU. We checked the amount of work waiting in the run queues, and the number of online CPU's:

```
for_each_cpu_and(cpu, sg_cpus, env->cpus) {
    if (cpu_online(cpu))
    {
        run_qu_tot+=cpu_rq(cpu)->nr_running;
        online_cpu+=1;
    }
```

Figure 10- counting the run queue and online CPU's

After the loop, we add another check to see if there is an imbalance according to the new average.

```
avg_load = run_qu_tot/online_cpu;
if (env->dst_rq->nr_running > avg_load)
    return 1;
```

Figure 11- checking if there is an imbalance

Find busiest group

This function starts by updating the Sched domain load balance statistics, after the update we have the local and busiest Sched group statistics to work with:

```
static struct sched_group *find_busiest_group(struct lb_env *env)
{
    struct sg_lb_stats *local, *busiest;
    struct sd_lb_stats sds;
    init_sd_lb_stats(&sds);

    /*
     * Compute the various statistics relavent for load balancing at
     * this level.
     */
    update_sd_lb_stats(env, &sds);
    local = &sds.local_stat;
    busiest = &sds.busiest_stat;
```

Figure 12-find busiest group start

From this point we have many tests to see if we found the group or not, to them we added a new test:

```
if (local->group_new_avg < busiest->group_new_avg)
    goto force_balance;
```

Figure 13- the added code to find busiest group

As we see, if we find a group that has a bigger average than our group, we will go to force balance, which calculates the imbalance- the amount of tasks to move in order to have a balanced system.

Calculate imbalance

Here we added a new calculation for the amount of jobs to be moved, the calculation is based on the total amount of jobs and the online CPU's:

```
if(local->group_new_avg < busiest->group_new_avg)
{
    int onlineCpus= local->number_of_working_cpus +  busiest->number_of_working_cpus;
    int tasksCount = local->sum_nr_running + busiest->sum_nr_running;
    int neededAvg = tasksCount/onlineCpus;
    int tasksDiff = max(0,busiest->group_new_avg - local->group_new_avg);
    env->imbalance=max(0,busiest->group_new_avg-neededAvg);

    return;
}
```

Figure 14-the new imbalance calculation

Move tasks

As explained in previous chapters, in this function we had to disregard the calculation result from the *task_h_load* function, as can be seen in the next figure.

```
load = task_h_load(p);

if (sched_feat(LB_MIN) && load < 16 && !env->sd->nr_balance_failed)
{
    goto next;
}

if ((load / 2) > env->imbalance)
{
        //goto next;
}
```

Figure 15 - commented code from the move tasks function

Important remark: we did not change anything in the load balance function itself, but only in its helper functions.

Chapter 4: Evaluation & Comparison

Overview

This chapter shows the results after all the changes and compares them to output we had from chapter-1.

Experiment Setup

We installed the fixed Kernel on our system, and then we executed the tool that was created on chapter-1. We also created a test to see how the system behaves when one CPU is down, the system has tasks waiting for execution and we add a new task.

Evaluation

The first case we will examine is when the system is busy and we shut down one CPU, the following graph shows the results:

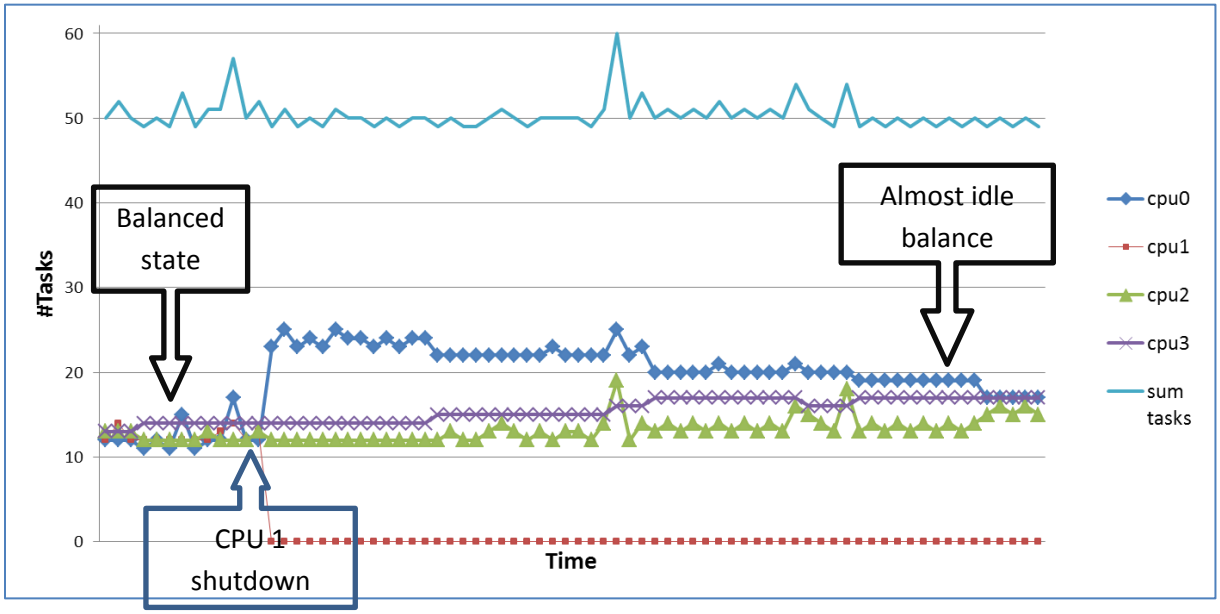


Figure 16- system state after the fix

At the end the system is in an almost idle balance state, which tells us that the solution is working.

We start in a balanced stated, when all the 4 CPU's are working, at the point of CPU 1 shout down, we have the same behavior as in chapter 1, when all of the CPU tasks are moved to CPU 0. Then with some time we can detected that tasks are starting to move from the CPU and to the other two CPU's that are also working. At the end of the graph we have a balanced system when each CPU has almost the same amount of job as the others.

We can also see in Figure 10 the total amount of tasks in the system, this line is to show that the amount of jobs in the system is almost the same and that the balancing is caused by our changes and not because of something else (like tasks leaving the system).

Now let's look at the second test, the system is busy and we are adding a new task to the system:

```
in get_cpu_workload-round 118:cpu num is 1 and number of proc is 17
in get_cpu_workload-round 118:cpu num is 2 and number of proc is 0
in get_cpu_workload-round 118:cpu num is 3 and number of proc is 15
in get_cpu_workload-round 118:cpu num is 4 and number of proc is 17
```

Figure 17 - system state before adding a new task

In the figure we see that the system is balanced, group 1 (CPU's 1 and 2) has an average of 17 while group 2(CPU's 3 and 4) has an average of 16.

When a new task would be added, the groups' average is checked and we expect the new task to be added to group 2:

```
in get_cpu_workload-round 119:cpu num is 1 and number of proc is 17
in get_cpu_workload-round 119:cpu num is 2 and number of proc is 0
in get_cpu_workload-round 119:cpu num is 3 and number of proc is 16
in get_cpu_workload-round 119:cpu num is 4 and number of proc is 17
```

Figure 18-system state after adding a new task

As we a new task was added and it was added to group 2, as we expected.

Chapter 5: Conclusions & Additional Info

Conclusions

We started with a difficult and very interesting mission, to fix a deep and unknown issue in the Linux kernel.

We have seen that by changing a few functions in the load balancing mechanism we were able to achieve proper load balancing in a case of offline CPU, which is not supported in the current Kernel versions.

The whole project was done on v3.14, in the newer versions the load balancing section is being remodeled and improved but this problem still exists.

Additional Information

Source code

The source files to our project can be located on:

1. The webpage of the Software Systems Lab of the Department of Electrical Engineering of the Technion – Israel Institute of Technology at the below provided address, under the name of this project.
   http://nssl.eew.technion.ac.il/recently-completed-projects/
2. Github- Improving-Linux-Load-Balancing-project

Contact Information

Boaz Haim

- Mail : mailto:bhaim123@gmail.com
- LinkedIn - profile

Bibliography

[1] Bovet, D., & Cesati, M. (2005). Understanding the Linux kernel (3rd Ed.). Beijing: O'Reilly.

[2] Mauerer, W. (2008). Professional Linux kernel architecture. Indianapolis, IN: Wiley Pub.

[3] Linux Load Balancing Mechanisms-NTHU Institution-
http://nthur.lib.nthu.edu.tw/bitstream/987654321/6898/13/432012.pdf

[4] Kernel Documentation:
https://www.kernel.org/doc/Documentation/scheduler/sched-domains.txt

[5] Seeker, V. (2013). Process Scheduling in Linux. University of Edinburgh
http://criticalblue.com/news/wp-content/uploads/2013/12/linux_scheduler_notes_final.pdf

Appendix A – functions differences

Overview

This appendix will show the comparison between the major functions in the Kernel and the modified functions from the project. All the comparisons were done using BeyondCompare tool, the left side is the original code and the right side is the changed one. When a line is marked, it means that it is missing in the other function / data structure.

Calculate imbalance

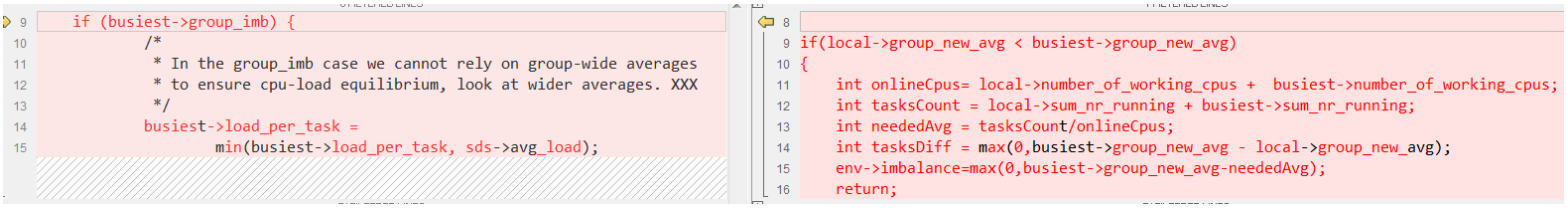


Figure 19 - calculate imbalance differences

Find busiest group

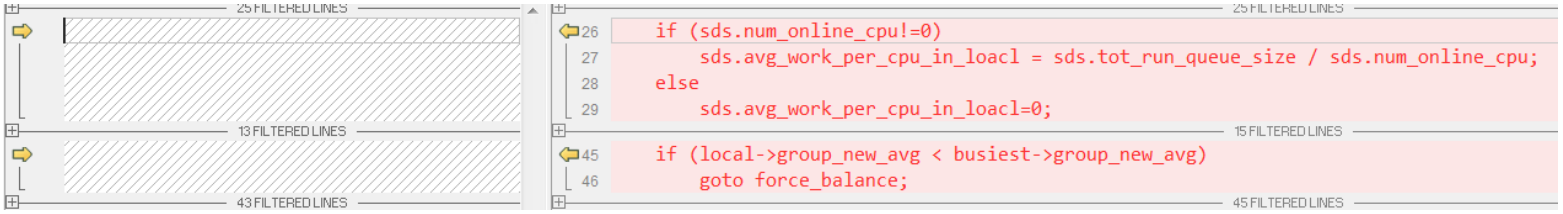


Figure 20 - find busiest group differences

Update sd lb stats

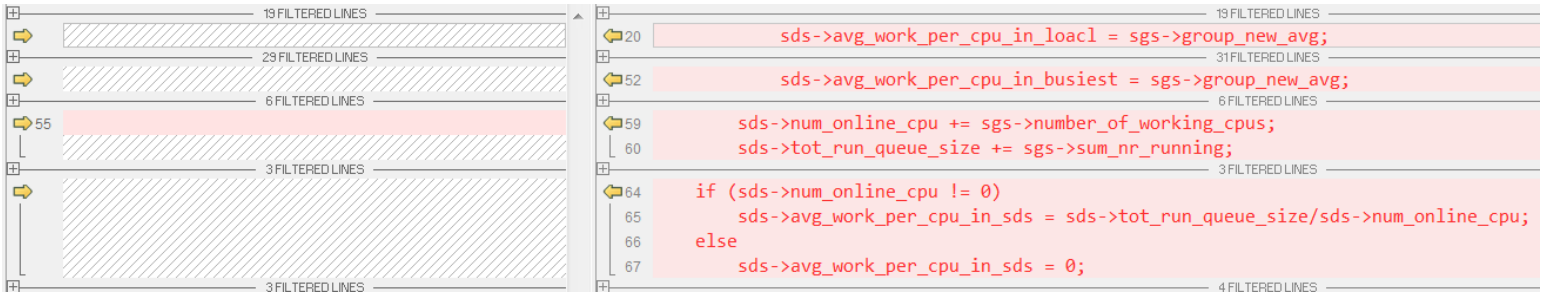
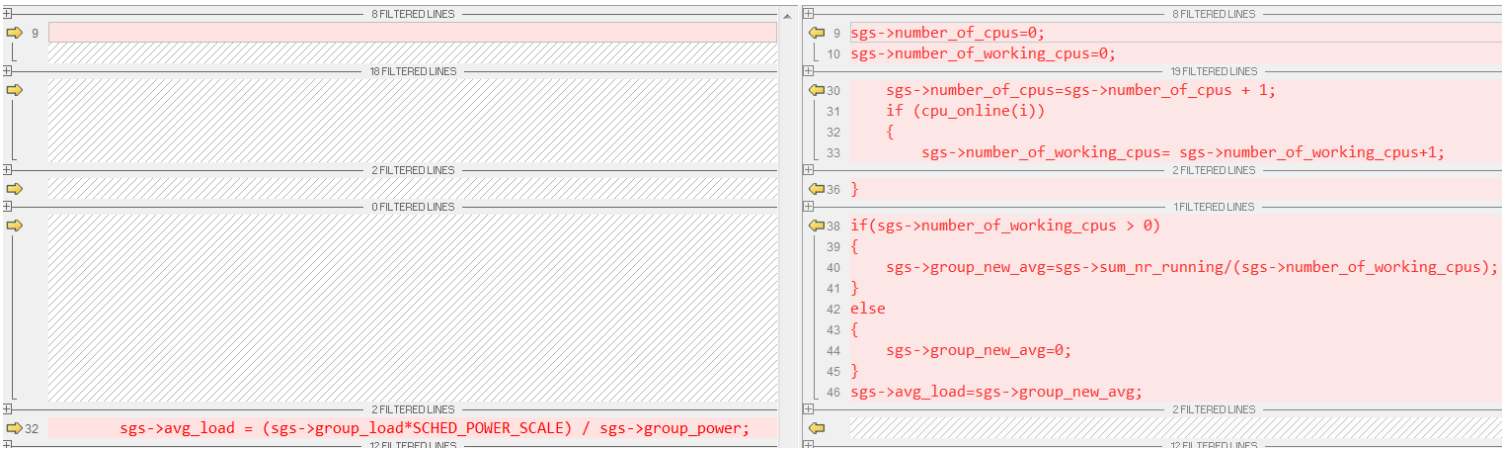


Figure 21 - Update sd lb stats differences

Update sg lb stats

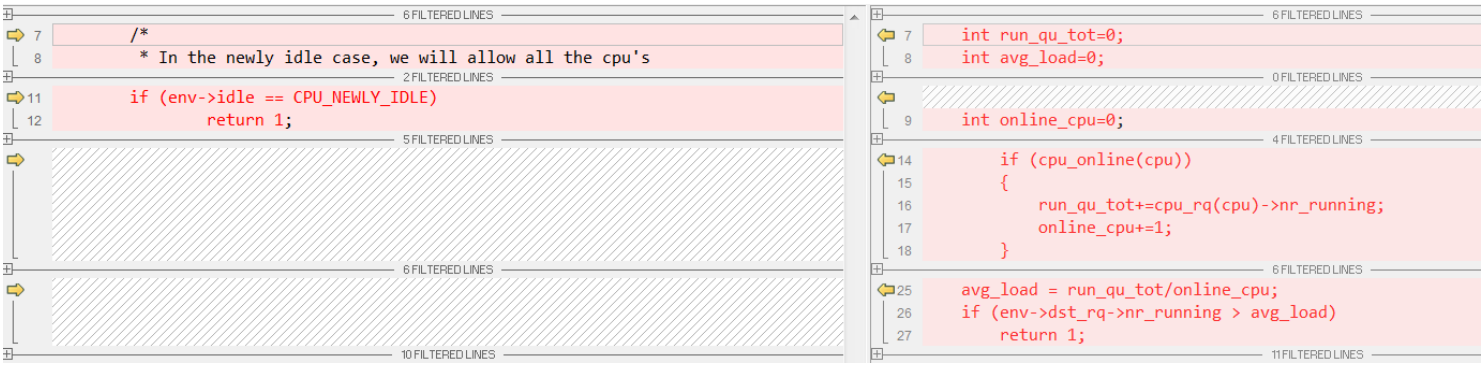


```
8 FILTERED LINES
9
18 FILTERED LINES
2 FILTERED LINES
0 FILTERED LINES
2 FILTERED LINES
32      sgs->avg_load = (sgs->group_load*SCHED_POWER_SCALE) / sgs->group_power;
12 FILTERED LINES
```

```
8 FILTERED LINES
9  sgs->number_of_cpus=0;
10 sgs->number_of_working_cpus=0;
19 FILTERED LINES
30     sgs->number_of_cpus=sgs->number_of_cpus + 1;
31     if (cpu_online(i))
32     {
33         sgs->number_of_working_cpus= sgs->number_of_working_cpus+1;
2 FILTERED LINES
36 }
1 FILTERED LINES
38 if(sgs->number_of_working_cpus > 0)
39 {
40     sgs->group_new_avg=sgs->sum_nr_running/(sgs->number_of_working_cpus);
41 }
42 else
43 {
44     sgs->group_new_avg=0;
45 }
46 sgs->avg_load=sgs->group_new_avg;
2 FILTERED LINES
12 FILTERED LINES
```

Figure 22 - Update sg lb stats differences

Should we balance



```
6 FILTERED LINES
7       /*
8        * In the newly idle case, we will allow all the cpu's
2 FILTERED LINES
11      if (env->idle == CPU_NEWLY_IDLE)
12              return 1;
5 FILTERED LINES
6 FILTERED LINES
10 FILTERED LINES
```

```
6 FILTERED LINES
7   int run_qu_tot=0;
8   int avg_load=0;
0 FILTERED LINES
9   int online_cpu=0;
4 FILTERED LINES
14      if (cpu_online(cpu))
15      {
16          run_qu_tot+=cpu_rq(cpu)->nr_running;
17          online_cpu+=1;
18      }
6 FILTERED LINES
25  avg_load = run_qu_tot/online_cpu;
26  if (env->dst_rq->nr_running > avg_load)
27      return 1;
11 FILTERED LINES
```

Figure 23 -Should we balance differences