POWER EFFICIENT SCHEDULING FOR A CLOUD SYSTEM

Joachim Sjöblom

Master of Science Thesis
Supervisor: Prof. Johan Lilius
Advisor: Dr. Sébastien Lafond
Embedded Systems Laboratory
Department of Information Technologies
Åbo Akademi University
September 2011
ABSTRACT

The aim of this thesis is to investigate the Linux Kernel and evaluate the available power-saving functionality. We propose changes to the current scheduler to improve said power-saving and explaining the role said improved scheduler would have in a Cloud System managed by a PID controller. We simulate scheduling for a number of different topologies and workloads using LinSched. This thesis presents results gathered for both the proposed scheduler and the current Linux scheduler and doing a comparison. The results indicate that the proposed scheduler has potential, but needs further work.

Keywords: Cloud Computing, Completely Fair Scheduler, Linux, Power
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALB</td>
<td>Active Load Balance</td>
</tr>
<tr>
<td>API</td>
<td>Application Programmming Interface.</td>
</tr>
<tr>
<td>CFS</td>
<td>Completely Fair Scheduler.</td>
</tr>
<tr>
<td>CPU</td>
<td>Central processing unit.</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In, First Out.</td>
</tr>
<tr>
<td>HOS</td>
<td>Host Operating System.</td>
</tr>
<tr>
<td>ILB</td>
<td>Idle Load Balance/Balancer</td>
</tr>
<tr>
<td>LB</td>
<td>Load Balancing or Load Balancer.</td>
</tr>
<tr>
<td>OGS</td>
<td>Overlord Guided Scheduler.</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional-Integral-Derivative.</td>
</tr>
<tr>
<td>PM</td>
<td>Power Manager</td>
</tr>
<tr>
<td>RBT</td>
<td>Red-Black Tree.</td>
</tr>
<tr>
<td>RQ</td>
<td>Runqueue.</td>
</tr>
<tr>
<td>RR</td>
<td>Round Robin.</td>
</tr>
<tr>
<td>RT</td>
<td>Real-Time.</td>
</tr>
<tr>
<td>RTS</td>
<td>Real-Time Scheduling.</td>
</tr>
<tr>
<td>SD</td>
<td>Scheduling Domain.</td>
</tr>
<tr>
<td>SG</td>
<td>Scheduling Group.</td>
</tr>
</tbody>
</table>
SMP Symmetric Multiprocessing.

vruntime Virtual Runtime.
GLOSSARY

**ALB**  A more aggressive type of LB. Requires at least one task per physical CPU and will push running tasks off the busiest RQ to accomplish this.

**Busy Time**  Period of time during which the CPU is executing.

**Cloud**  Platform for computational and data access services where details of physical location of the hardware is not necessarily of concern for the end user.

**CPU**  Will herein be used colloquially for both logical and physical CPUs, as well as cores. If a distinction is to be made then the whole term of the resource in question will be used.

**Data center**  Facility that houses computer systems.

**DVFS**  Dynamic Voltage and Frequency Scaling is used to adjust the input voltage and clock frequency according to the momentary need in order to avoid unnecessary energy consumption.

**Granularity**  Granularity describes the extent a system is broken down into smaller parts.

**HZ**  Constant defining the timer interrupt frequency in the Linux Kernel.

**ILB**  Used in tickless scheduling. Idle CPUs have their ticks turned off and will thus, if they are needed, have to be reactivated by the non-idle CPU that has been designated as the ILB.

**Idle Time**  Period of time during which the CPU is completely idle.

**Jiffies**  Variable which stores the amount of passed time quantum - normally one quantum is 10ms - since system bootup.
**Scheduling Domain** A scheduling domain is a set of CPUs which share properties and scheduling policies. Scheduling domains are hierarchical; a multi-level system will have multiple levels of domains.

**Scheduling Group** Each domain is split into scheduling groups. E.g. in a uniprocessor or SMP system, each physical CPU is a group.

**Server farm** A server farm is a collection of servers. They are used when a single server is not capable of providing the required service.

**SMP** In symmetric multiprocessing two or more processors are connected to the same main memory.

**Tickless Scheduling** Only use periodic ticks when a CPU is not idle.

**vruntime** The variable storing the amount of time a CFS task has spent on a CPU.
LIST OF FIGURES

2.1 Example of a red-black tree [14] ........................................... 6
2.2 Structure hierarchy for tasks and the red-black tree [14] ........ 7
2.3 Illustrating the CFS RQ selection process. .............................. 8
2.4 SCHED_MC not in use [13] ................................................. 13
2.5 SCHED_MC in use [13] ....................................................... 13
2.6 SCHED_SMT not in use [13] ................................................. 14
2.7 SCHED_SMT in use [13] ....................................................... 15
2.8 Example of the Load Averages over a 24 hour period. [9] ....... 19

3.1 A very simple example of a PID controller. ............................. 21

4.1 Illustration of how the weighted CPU load is assumed to affect perceived business ......................................................... 34


6.1 Execution and delay times for a Dual Socket Quad CPU .......... 51
6.2 CFS and OGS Execution times ratioed for a Dual Socket Quad CPU ................................................................. 52
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Abbreviations</td>
<td>ii</td>
</tr>
<tr>
<td>Glossary</td>
<td>iv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>Contents</td>
<td>viii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>1.1 Cloud Software project</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Purpose of this thesis</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Thesis Structure</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Scheduling and Power Management in the current Linux Kernel</strong></td>
<td>4</td>
</tr>
<tr>
<td>2.1 The Completely Fair Scheduler</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 CFS Runqueue</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2 Runqueue Selection</td>
<td>6</td>
</tr>
<tr>
<td>2.1.3 Load Balancing</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Governors</td>
<td>16</td>
</tr>
</tbody>
</table>
2.3 CPU Load Average ........................................... 17

3 Controlled Power Management .................................. 20
3.1 Basic PID Control Theory ..................................... 20
3.2 Controlling a Web Cluster using a PID ..................... 22

4 Proposed Scheduler Changes ................................. 24
4.1 Runqueue Selection ........................................... 25
4.1.1 find_idlest_group() vs find_busy_group() .......... 25
4.1.2 find_idlest_cpu() vs find_busy_sched_cpu() ..... 29
4.1.3 what_is_idle_time() ...................................... 31
4.2 Load Balancing ............................................... 34
4.2.1 find_busiest_group() ..................................... 35
4.2.2 find_busiest_queue() ..................................... 37
4.2.3 calculate_imbalance() ................................... 40
4.2.4 move_tasks() ............................................. 42

5 Tools, Experimentation and Testing ......................... 44
5.1 Linsched ....................................................... 44
5.1.1 LinSched Tasks .......................................... 45
5.1.2 Simulator Limitations .................................... 46
5.1.3 Workarounds ............................................. 46
5.2 Simulated Workloads ........................................ 47
5.3 Power Consumption .......................................... 49

6 Results ........................................................... 50

7 Further Work .................................................... 63

8 Conclusion ......................................................... 65

ix
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bibliography</td>
<td>67</td>
</tr>
<tr>
<td>Swedish Summary</td>
<td>69</td>
</tr>
<tr>
<td>A Linux Kernel Change Log</td>
<td>76</td>
</tr>
<tr>
<td>B LinSched Change Log</td>
<td>86</td>
</tr>
</tbody>
</table>
CHAPTER
ONE

INTRODUCTION

A rapidly growing population, both online and offline, is creating an ever increasing demand for electricity. At the same time, an increasing climate awareness has caused a demand for higher and higher power efficiency in, among other things, computers. A lower power consumption is also more economical.

With the Internet and networking slowly becoming ubiquitous server clusters receive more and more requests, which means they have to be expanded in order to handle the increase. The expansion again will lead to more power being consumed, both due to there being more electronics to be powered and because said electronics require cooling and ventilation. Solving this problem simply by increasing the amount of servers in the farms is not sustainable in the long run, as said farms dissipate a sizeable amount of power even when not under heavy loads [1] [10]. Currently these server centres, generally speaking, consume more power than they have to. This is partly because the power consumption scaling is not proportional to the current load over all the boards and processors in the system [17]. That is to say, they have no way of turning parts of the system off if the part in question, based on the systemwide load, is not currently needed. This is where this thesis comes in.

Another student at Åbo Akademi University recently proposed a system where a Proportional-Integral-Derivative (PID) Controller would be used to calculate and control the number of processors (CPU) needed to meet the current demand on the system [11]. While this thesis does not actually concern itself with control theory, there is however a relationship between the
aforementioned controller and the performed work. This relationship as well as some basic controller theory will be explained in more detail in Chapter 3.

At the time of writing it is quite common to use components engineered specifically for server systems. These components are thus built to be able to process a sizeable number of tasks which, since CPU power is proportional to the amount of energy dissipated, means they require quite a lot of energy even when not fully loaded. The proposed solution would utilise CPUs from the ARM family instead of the conventional server-grade CPUs. The ARM CPUs have been engineered with smartphones and embedded systems, which are both environments with inherently scarce resources, in mind. Subsequently their performance is not as good as that of a server-grade CPU, but on the other hand they require a lot less in terms of power. Thus using a large amount of ARM CPUs would give us a better granularity in terms of energy dissipated versus demand for CPU power [21]. If the system’s energy dissipation matches the demand for CPU power then we have a very energy efficient system [19].

1.1 Cloud Software project

The Cloud Software Program (2010-2013) is a SHOK-program financed through TEKES and coordinated by Tivit Oy. Its aim is to improve the competitive position of the Finnish software intensive industry in the global market. The content of this thesis is part of the project.

The research focus for the project in the Embedded Systems Laboratory at The Department of Information Technologies at Åbo Akademi is to evaluate the potential gain for energy efficiency by using low power nodes to provide services. In addition to energy efficiency the total cost of ownership for the cloud server infrastructure is in a central role.

1.2 Purpose of this thesis

This thesis will investigate the Linux kernel’s functionality in terms of scheduling and power-saving. The available functionality’s suitability for use in a Cloud Service with Controlled Power Management will be evaluated and some potential changes will be proposed.

The evaluation will herein be based on simulation results provided by Linsched which are visualised and analysed in Excel, as well as available kernel documentation.
1.3 Thesis Structure

Chapter 2 will cover the scheduling and power-saving functionality currently available in the Linux kernel starting with the Completely Fair Scheduler (CFS) and its design. This is then followed by a quick look at Real-Time Scheduling (RTS) and Load Balancing (LB). Lastly this chapter will cover the Governors, their purpose and how they accomplish it. Chapter 3 explains how the aforementioned Controlled Power Management is designed and will also clarify where this thesis fits in. Chapter 4 will explain how the kernel could be changed to improve the results achieved with the Power Management (PM), starting with Runqueue (RQ) selection. The LB is not quite ideal either and related change proposals will be mentioned next. As the PM will use a feedback loop, we will need the kernel to report a metric to the Controller and a few ways of accomplishing this will be explained next. The PM will also let the system know how many CPUs are needed to meet the current demand and thus the last part of this chapter will be a look at how this could be done. In chapter 5 we will look at the tools that were used for evaluating performance, as well as how the tests were constructed. Chapter 6 will cover the results and conclusion, while chapter 7 will list potential further work.
One obvious place to start, when investigating the kernel’s scheduling and power-saving functionality, is the available schedulers. We are not currently interested in older versions of the kernel, so the old O(1) scheduler will not be taken into consideration, hence the focus on the Completely Fair Scheduler (CFS).

2.1 The Completely Fair Scheduler

While the O(1) scheduler solved many of the problems plaguing the pre-2.6 kernels - it eliminated the need to iterate through the entire task list to identify the next task to schedule, which vastly improved scalability and efficiency - it did require a large mass of code to calculate heuristics and was difficult to manage. These issues and other external pressures led to Ingo Molnar developing the CFS, which he based around some previous work by Con Kolivas [14]. CFS has also done away with the time slices which were used by previous Linux schedulers [16].
2.1.1 CFS Runqueue

Like its name implies, the main purpose of the CFS is to give all tasks a fair and balanced amount of CPU time. To determine the amount of time each task should be allocated, the CFS keeps track of the amount of time each task has already spent on a CPU in a per-task variable called virtual runtime. Thus the smaller the virtual runtime, the less time a task has spent on a CPU and, subsequently, the higher its need for CPU time. CFS is also capable of ensuring that tasks that are not currently runnable, e.g. tasks waiting for user input, will receive a comparable share of CPU time when they actually need it [14].

CFS does not use the same type of queue structure the previous Linux schedulers have though, but rather maintains a time-ordered red-black tree (RBT). RBTs are self-balancing, which means that there is never a path in the tree that is more than twice as long as any other, and operations on the tree occur in $O(\log(n))$ time, where $n$ is the number of nodes in the tree. This means that tasks can be inserted or deleted quickly and efficiently. Figure 2.1 below is an example of an RBT as it may look when used in CFS. The further to the left in the tree a task is stored, the greater its need for CPU time. Conversely, the further to the right, the lesser the need. To maintain the aforementioned CPU time balance CFS chooses the leftmost task as the next task to be scheduled. When the task is done, it adds the time it spent on the CPU to the virtual runtime and, if still runnable, is then inserted back into the tree. The self-ordering attribute combined with the scheduler always updating the virtual runtime according to the amount of time spent on a CPU and always choosing the leftmost task in the tree will ensure we have a balanced CPU time across the whole set of runnable tasks [14].

All tasks in Linux are represented by a task structure called `task_struct`, wherein the tasks’ current state, its stack, process flags, static and dynamic priorities are stored. This structure does not include anything CFS-related however, so with the introduction of CFS a new structure for tracking scheduling information called `sched_entity` was also introduced. Hierarchically speaking, as can be seen in Figure 2.2, the `task_struct` sits on top and fully describes the task. It includes the `sched_entity` structure. The `sched_entity` then, as mentioned, stores CFS-related scheduling information and what is probably the most important element; the vruntime. It serves to keep track of the time a task has spent on a CPU and also, by extension, as the RBT index. On the next level down, we then find the `cfs_rq` struct of which there is one per RQ and holds information about its associated RBT [14] [16].
2.1.2 Runqueue Selection

When a new task is created, the scheduler has to select a RQ onto which to schedule it. This is done by calling the `select_task_rq` function in `sched.c`. The function call is actually scheduling class-specific and the algorithm that is actually used to select the RQ will depend on the task’s type. Tasks classified as RT tasks will thus use a different algorithm from the ones that are not. We are currently not interested in RT tasks however, and will focus on the task classification handled by the CFS task selection algorithm [5].

The `select_task_rq` call in `sched.c` will call a function with the same name located in the file `sched_fair.c`, where the concrete RQ selection decisions are made. We begin by iterating through the scheduling domains looking for a suitable target domain. The chosen target domain will be the lowest domain fulfilling all the selection criteria - i.e. the last domain in the for-loop to fulfill said criteria. The aforementioned criteria are dependent on the flags set for the current scheduling domain and will thus change slightly depending on SD conditions outside this function’s control [6].

As is illustrated in Figure 2.3, once a target SD has been found, we enter a while-loop and start out by double-checking the suitability flags. If the checks fail we move to a lower domain and try again. When the checks
have been cleared we look for the idlest group in the domain by calling
\texttt{find\_idlest\_group()}. This function is quite simple and essentially just tallies
the load for each scheduling group (SG) in the current domain, selects
the idlest group and returns it. If \texttt{NULL} is returned, i.e. if no suitable SG was
found, we move on to a lower SD and try again. When a SG is returned we
then want to find the idlest CPU, which is done by calling \texttt{find\_idlest\_cpu()}. This function is also quite simple, as it just iterates through the CPUs in the SG
and selects the one with the lowest \texttt{weighted\_cpuload}, which it then returns.
Again, if no suitable candidate is returned, we move on to a lower SD and try
again from the top of the while-loop. Then, before returning the CPU onto
which the new task is to be scheduled, we do one last thing, namely make sure
there is no lower level on the CPU which would be a more suitable candidate.
If we do find one, then we iterate through the while-loop again. If we don’t;
the while-loop breaks, the chosen CPU is returned and we are done. This being
the core of CFS’ RQ selection thus means that all CPUs will utilised equally,
which, when considering how we would like to save power, is not ideal. The

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{structure_hierarchy.png}
\caption{Structure hierarchy for tasks and the red-black tree [14]}
\end{figure}
functionality is actually the exact opposite of our intentions [6].

![Diagram of CFS RQ selection process]

**Figure 2.3:** Illustrating the CFS RQ selection process.

### 2.1.3 Load Balancing

Since the set of runnable tasks is not constant while the system is running, the conditions for an ideal task distribution are bound to change. When it does we will need some way of redefining the task distribution, which is where the load balancer (LB) comes in. Naturally the use of a load balancer is redundant for systems with only one CPU.

The Linux scheduler is built to tick periodically, with the length of the tick period being dependent on the $HZ$ value used. The $HZ$ defines how frequently we get a timer interrupt. Traditionally, up until kernel version 2.4, the $HZ$ value has been 100, which equals one interrupt every 10ms. Nowadays
the default value is on the order of 1000, which equates to an interrupt interval of 1ms [8]. On every interrupt the timer code calls `scheduler_tick()` in `sched.c`, at which point we perform some scheduling-related maintenance. We will focus on the parts related to LB, since the other parts are not relevant for what we want to accomplish. As mentioned before, LB is not necessary if we are not using an SMP system. In the CFS case the LB process begins with `trigger_load_balance()` in `sched_fair.c` being called from `scheduler_tick()` in `sched.c`.

**Tickless Scheduling**

Linux has since at least Kernel version 2.6.21 been able to do tickless scheduling. Using tickless will currently mean the disabling of periodic ticks for idle CPUs [18].

Normally, the CPU will receive a periodic interrupt, which the scheduler can use as a wake-up call and move from having been idle if need be. Consequently all CPUs in the system will be woken up or interrupted periodically, even if they are idle. This is not ideal, as idle CPUs should be allowed to stay idle for as long as they are not needed, hence the concept of tickless scheduling. With the interrupts, which are essentially the same as ticks in this context, turned off a CPU will be able to stay idle for a lot longer. This scheme does mean the idle CPUs have no way of waking up on their own, which is a problem. The solution, since it is relatively closely related to LB, will be explained to some extent in the following section [18].

**The Load Balancing Procedure**

1. `scheduler_tick()` in `sched.c` is called periodically by the timer code as mentioned above.

2. `trigger_load_balance()` in `sched_fair.c` is called from #1. The functionality herein depends on whether or not we are using tickless scheduling. Having to concern ourselves with whether or not we use tickless scheduling when the function call which brought us into this function depends on ticks may sound counter-intuitive. This is simply due to tickless’ having turned off interrupts for idle CPUs, and their need for being looked after. If we are indeed using tickless scheduling, then idle CPUs rely on a specific and herein defined CPU, called ”idle load balancer” (ILB), for their LB. At the top of this function we do a check to see if the ILB CPU should change. We also make sure the CPU is not idle,
as we may not need it and thus return if we do find it to be idle. If we pass all these checks, or if we simply are not using tickless scheduling, we check to see if it is time to perform LB for this RQ and CPU. If it is, then a soft IRQ is raised after which the next function in the process is called.

3. \textit{run\_rebalance\_domains()} in \textit{sched\_fair.c} is called from \#2. This function’s functionality is also, to a degree, dependent on whether or not we use tickless scheduling. First off we will do LB for the current CPU. However, this function does not really do anything in that regard. We will simply find out the CPU ID and get the RQ and idle type of the CPU in question, after which \textit{rebalance\_domains()} is called. When we return from that function - this is either when the LB is finished, or if we fail one of the checks on the way - we then move on to do ILB for the idle CPUs if this CPU is responsible for it.

4. \textit{rebalance\_domains()} in \textit{sched\_fair.c} is called from \#3. This function will check each SD to see if it is due to be balanced and will initiate balancing procedure. Thus most of what this function does is to calculate how long it’s been since the last LB and check if we have exceeded the threshold value. It will also verify the need for serialisation, or lack thereof. We then call \textit{load\_balance()} for the CPUs and SD for which it is required.

5. \textit{load\_balance()} in \textit{sched\_fair.c} is called from \#4. Here is where we actually start looking at how balanced the CPU is within its domain. Quite a bit of the functionality here is related to an obscure branch of the scheduling statistics which do not seem to have any impact on the actual LB. The parts we are interested in are \textit{find\_busiest\_group()}, \textit{find\_busiest\_queue()} and \textit{move\_tasks()}, which are called in that order.

6. \textit{find\_busiest\_group()} is called from \#5. The result is returned to \#5. It does pretty much what the name implies. It finds the busiest group in the SD. To start with we need to make sure we have up to date statistics to work with. These are not the same statistics as in \#5, but rather a set that lets us know which SG is busiest, which SG is idlest as well as overall SD load and power. Next we perform a number of checks, which, if true, tell us there is no imbalance from the point of view of the current CPU and we jump to \textit{out\_balanced}. These cases are quite logical and intuitive. We will explain these in further detail in Chapter 4.2. If none of the cases apply we move on to \textit{calculate\_imbalance()}, which does what the name implies. Based on the average load per task in the SG, it calculates the amount of weighted load we have to move in order to equalise the perceived imbalance and stores the value. This value is
stored in `imbalance` before returning to `find_busiest_group`, which then returns the busiest SG to #5.

7. `find_busiest_queue()` is called from #5 using the result from #6. The result is returned to #5. Before this function is called, we will make sure that the value returned from #6 actually points to a SG. Provided it does, we then proceed to look for the busiest RQ in the SG. This is quite simply done by iterating through all the RQs in the SG and returning the one with the highest weighted load. The RQ is then returned to #5.

8. `move_tasks()` is called from #5 using the information given by #6 and #7. If the busiest RQ has only one task running, then attempting to move it is pointless and this step is skipped. Otherwise both the target and the busiest RQs will be locked and we attempt to equalise the imbalance by pulling tasks to the target RQ. The actual pulling is done through another sequence of calls starting from this function by calling `load_balance_fair()`, but they only make sure we move tasks that are actually moveable and thus do not affect the choice of RQ and SG. Due to this they are not of any immediate interest here. We then iterate the do-while loop in `move_tasks()`, with each iteration reporting the amount of weighted load that was moved, until we have moved the amount specified in `imbalance` by #6.

When #8 is finished it returns to #5, where the previously locked RQs are now unlocked and we then proceed to check whether or not the move was successful. If it was not then we check whether or not it was because all tasks were pinned - pinned tasks are flagged during task pulling - or if it is because active load balancing (ALB) is required. ALB is the more aggressive and less often used alternative to LB. It requires at least one task to be running on each physical CPU where possible and will pull running tasks off the busiest CPU in order to accomplish this. Other than being more aggressive, only moving one task to an idle CPU and looking at different flags, it does not really differ from normal LB [7].

If we did not need to perform ALB and simply failed to move tasks, then a negative value is returned to #4. The only thing the return value affects at this point is whether or not the CPU’s idle type is changed. If we successfully moved one or more tasks, then the target CPU is obviously no longer idle and should be flagged as such. If we were unsuccessful, then we leave the target CPU flagged as being in the state it was at the beginning of the LB procedure. We then proceed to update the time when the next LB is due for the current SD. With this the LB procedure is finished.
The end result of the CFS’ default LB procedure, detailed above, is normally an even task distribution over the CPUs available in the system. Therefore using the default CFS will not save us any power and as such it is unsuitable for our purposes.

The CONFIG_SCHED_MC Setting

As was mentioned previously; when used in its default mode, the CFS will attempt to essentially optimise performance by distributing tasks equally over all available CPUs. While this, combined with the next task being chosen based on its need for CPU time, does help overall performance, it does nothing to improve power-saving. Attempting to use all resources equally is essentially the opposite of what we are trying to achieve. This, however, does not mean that CFS is completely incapable of saving power, just that it by default does not.

When building a kernel we are given a large number of options so as to let us tailor the build to our needs. One of these options is called SCHED_MC and appears in the Processor Type and Features section of the Kconfig file. This means that the option has to be chosen at build-time in order for the setting to be included and useable. To paraphrase the help print in the Kconfig file it is used for: Multi-core scheduler support improving the CPU scheduler’s decision making when dealing with multi-core CPU chips at a cost of slightly increased overhead in some places [2]. What SCHED_MC then does is to attempt to consolidate our running tasks to as few cores as possible in order to save power [13].

Figure ?? shows the utilisation rate of a system with 8 CPUs with SCHED_MC turned off. We can see that the CPUs are on average utilised to 10%. All the CPUs are used to some extent here even though it is clear we could make do with only a subset of them. Figure ?? is an example of the same system and the same conditions, but this time with SCHED_MC turned on. Now we can see that we only use half of the available CPUs in the system and have let the other half go idle. Although based on the Kernel code, and Figures ?? and ??, the SCHED_MC setting only seems to work for relatively low loads and only in a few cases. By selecting the appropriate option during build-time, we activate the functionality of a number of scheduling-related functions. Apart from adding elements necessary to keep track of the busiest SGs, we also activate the functionality in check_power_save_busiest_group(), which is called from find_busiest_group() - i.e. #6 in the LB procedure covered above - in four of the five cases when no imbalance exists from the point of view of the current
CPU. It is not called if the current CPU is not the appropriate CPU to perform LB at this level. The call to `check_power_save_busiest_group()` then results in the idlest SG being returned as the busiest, which, as it will then be used in the normal LB procedure as the busiest group, will result in our ending up pulling tasks from a RQ with a low load. In short then; by using the idlest SG’s busiest RQ and pull tasks from there to the target RQ, we will in the long run siphon off all tasks from barely loaded CPUs and can then let them go idle as was
seen in Figure ?? [13]. It is however unclear as to how well this works, and due to the non-existent load weight documentation it is also, due to the tight time constraints of this thesis, difficult to design any improvements. We will attempt to measure SCHED_MC’s performance using LinSched. The results will be covered in Chapter 6.

The CONFIG_SCHED_SMT Setting

The CONFIG_SCHED_SMT setting is quite similar to the SCHED_MC, but where SCHED_MC concerns itself with tasks, the SCHED_SMT will attempt to consolidate hyperthreads onto as few CPUs as possible. The same conditions apply to its usage as for SCHED_MC. Namely that it has to be selected when the Kernel is built in order for it to be included. It appears just before SCHED_MC in the same section of the Kconfig file. To quote the help print from Kconfig: SMT scheduler support improves the CPU scheduler’s decision making when dealing with Intel Pentium 4 chips with HyperThreading at a cost of slightly increased overhead in some places [2] [13].

Figure 2.6: SCHED_SMT not in use [13]
Figure 2.7: SCHED_SMT in use [13]

Figure ?? illustrates how a system with 16 CPUs might be utilised with the SCHED_SMT option turned off. Again we have an average utilisation of 10% at this stage. Then in figure ?? we have turned on the SCHED_SMT option and can now see - or should have been able to see, had the figure’s colours not been so poorly chosen - that almost all of the threads have been successfully consolidated to four of the CPUs. The other 12 are barely utilised. Using SCHED_SMT does not seem to have much of an impact on the scheduling code. Unlike SCHED_MC, and based on what we have been able to find in the Kernel code, it has the biggest impact on which SD flags are set and also plays a small role when choosing an ILB.

It is also in this case unclear how well the functionality performs and how to potentially improve on it due to lacking documentation. Both of these settings also seem to be exclusive for the X86 architectures, and since we intend to use ARM CPUs we will not be able to use either the SCHED_MC option or the SCHED_SMT option. They can, however, be used to measure power-saving performance against, as they seem to be the only energy efficiency-related scheduling functionality of currently available.
2.2 Governors

Another thing available in the current Kernels is the so-called CPUfreq subsystem. Since its introduction in the 2.6.0 Kernel it has made it possible to dynamically scale CPU frequencies. By making use of Governors to detect how much power a CPU actually needs and adjusting the frequencies to match, the subsystem can allow for increased power-saving with a negligible sacrifice in performance [12].

Like the `SCHED_MC` and `SCHED_SMT` settings, we have to include the subsystem at build-time to be able to use it. The choice will appear in the CPU Frequency scaling section of the `Kconfig`. If we choose to include the CPUfreq subsystem, we should also specify which of the five available Governors we wish to include. It is possible to include all five and switch between them depending on your need after the Kernel has been built. These five governors are [12]:

- The Performance Governor, which is statically set to the highest available frequency. The only available tunable here is what the Governor perceives as the maximum.

- The Powersave Governor, which is statically set to the lowest available frequency. It is essentially the inverse of the Performance Governor. Using it is normally not a good idea, as it can actually lead to an increased power consumption. All tasks take longer to complete at lower frequencies, so while keeping the frequency at a minimum will seemingly save us power, it will actually just make the CPU work slower, and may in the long run dissipate more power than it saves.

- The Userspace Governor, which lets us manually select and set frequencies. It is a slightly trickier Governor to use, as it bases all its decisions on what we as users have told it to do. Hence the name. It does have a number of daemons - essentially pre-made sets of rules - to make using it easier however. Although looking at this from another angle, we can see that this is the most flexible and customisable governor available. Where the other governors only react to and work with the current load, the Userspace Governor can be programmed to react to environmental changes. If you pull your laptop’s wall plug, the Userspace Governor can detect this and switch to a more aggressive power-saving policy for instance. This type of functionality is more complex than we need [12].

- The Ondemand Governor, which looks at the CPU utilisation and changes the frequency accordingly. If the utilisation is found to be less
than a threshold value, the governor steps down the frequency until the frequency which corresponds best to the current utilisation is found. When the utilisation exceeds the threshold value, the governor sets the highest available frequency. Using this governor lets us define the range of available frequencies, the governor’s sampling rate, as well as the utilisation threshold. This governor would be a good choice for power-saving. It is both autonomous, as opposed to the Userspace Governor, and the fastest to react to changes. However, jumping straight to the highest frequency when the threshold value is exceeded could lead to unnecessary spikes in power dissipation [12].

- The Conservative Governor is quite similar to the Ondemand Governor. The main difference between the two is that the Conservative Governor does not jump straight to the highest frequency when its threshold value is exceeded, but rather steps the frequency up gradually. This gives us both a finer granularity and fewer power dissipation spikes, although it does have a slight adverse effect on performance in some cases. If we quickly go from being barely utilised to fully loaded, then the Conservative Governor will iterate through all frequency steps before it gets to the highest frequency, for instance. Whereas the Ondemand Governor will reach the highest frequency in one leap and thus have less of an impact on performance. This governor would be another fine choice for our power-saving needs [12].

Based on attributes and descriptions, it would seem that our choice stands between the Ondemand and the Conservative Governor. It is impossible to say which one of the two strikes a better balance between power-saving and performance simply based on their descriptions however, so it would be most prudent to run extensive tests before making the ultimate decision. A governor would be recommended in either case, as it is a completely autonomous function, does not affect scheduling and gives us more a granular control over delivered and demanded frequency. Either way we look at it; using a governor will help us save power.

2.3 CPU Load Average

The CPU Load Average is a decaying moving average calculated in the Kernel to give us an idea of the average CPU Load over different intervals. From the shortest to the longest, we have a 1-minute, a 5-minute and a 15-minute average. They measure the trend in CPU utilisation, instead of taking snapshots like the CPU percentage, and include all demand for the CPU, not
only how much the CPU was active at the time of measurement. Their values should be taken in context of the number of CPUs in the system; meaning that a CPU Load Average of 1 is an ideal load for one CPU, where a Load Average of 2.7 is more than two, but less than three CPUs can handle.

The averages then illustrate how loaded the CPU has been over their respective intervals. Obviously the 1-minute average will be the most responsive and look quite spiky compared to the slower ones. Looking at this average, we should be able to pick out bursts in the CPU Load, although the spikes won’t be as pronounced as in a snapshot. The 5-minute average’s curve is already a lot smoother and a lot less sensitive to bursty loads. Where it may not be prudent to panic when the 1-minute average is showing an upward trend, an upward trend in the 5-minute average should be taken more seriously. As it works over a longer time, only an actual overall load increase will cause the 5-minute average to move upward. Similarly the 15-minute average’s pointing up is cause for concern. At least when we start approaching the upper limit of our system. The CPU Load has to increase quite steadily for the 15-minute average to react. As can be seen in Figure 2.8, the 15-minute average is quite steady over the whole measurement period, but the 1-minute average demonstrates quite dramatic fluctuations. The 5-minute average, albeit somewhat difficult to see in the figure, follows the 15-minute average more closely than its 1-minute counterpart. This can be interpreted as the system being quite capable of handling the loads it was subjected to over the measurement period. Had the 15-minute and 5-minute averages looked more like the 1-minute average, or if the longer averages had displayed plateaus at an arbitrary point greater than 1 over the measurement period, we could have concluded the average load was larger than one CPU in the monitored system could handle.

In short then; the Linux scheduler does not schedule tasks in a way that saves power. Turning the power-saving functionality, `SCHED_MC` and `SCHED_SMT`, on will help the situation somewhat, but it is unclear to what degree it helps, it will affect the LB rather than RQ selection for new tasks, and the documentation is lacking. We will thus have to measure the performance empirically. Also, the CPU Load Average will not inherently help us save any power, but it could assist the Controlled PM by acting as a feedback value. We will cover how in more detail in Chapter 3.
Figure 2.8: Example of the Load Averages over a 24 hour period. [9]
As was previously mentioned, a student here at Åbo Akademi University has suggested a power management scheme based on control theory, and more specifically PID controllers. While the work in this thesis does not concern itself with any actual control theory, it will be an important part of the actual implementation. Hence, to be able to accurately explain the role this work will play, it is of some import to briefly cover how PID controllers work. We will gloss over quite a few of the details, so the control theory covered here will be very basic, simply explaining what a PID controller consists of and what the different parts do. We will not cover any of the mathematics involved in making it work.

3.1 Basic PID Control Theory

PID control is a concept we normally associate with machinery, cruise control, or similar concepts where there is something concrete and obvious to actually control. In a cruise controller, the PID makes sure the car keeps a certain speed, whether it is going up a hill or down, by controlling the throttle. If we use the cruise controller as an example here, then it essentially accomplishes its task by feeding the difference between the expected speed (input) and the current speed (output) into the PID, which then calculates how much we potentially have to correct our speed by to achieve the target speed.
Figure 3.1: A very simple example of a PID controller.

Figure 3.1 shows us a simplified block diagram of the aforementioned PID controller setup. The input block is assumed to contain all sensors and hardware necessary to produce an input value, i.e. the target speed, from which we then subtract the feedback value, i.e. the current speed, to get the error value. The error value is then sent to the PID. The resulting equation might look something like:

$$e = r - y$$  \hspace{1cm} (3.1)

where $e$ (error) is the PID input value, $r$ is the target value, and $y$ is the output and feedback value. From the PID block we then get the sum of three calculations.

- **Proportional calculation**
  
  $$K_p e(t)$$  \hspace{1cm} (3.2)

  where $K_p$ is the proportional gain, and $e(t)$ is the error at time $t$. Increasing the $K_p$ value will result in a larger error value and therefore a more drastic correction. Conversely, if $K_p$ is small, we will see a less drastic correction to an error.

- **Integral calculation**
  
  $$K_i \int_{0}^{t} e(\tau) d\tau$$  \hspace{1cm} (3.3)

  where $K_i$ is the integral gain, and $e(\tau)$ the error integrated from time $0$ to now. The integral will contain the sum of all previous instantaneous errors and will therefore help us reach the target value quicker, provided we have chosen a proper $K_i$ value. If the value is too large, we will oscillate around the target value, and if it is too small, we will approach the target value very slowly.

- **Derivative calculation**
  
  $$K_d \frac{d}{dt} e(t)$$  \hspace{1cm} (3.4)

  where $K_d$ is the derivative gain, and $e(t)$ will be differentiated with respect to time. This calculation will help us slow the output’s rate of
change. A large $K_d$ means a slower change, where a smaller value means a quicker change. As opposed to the proportional calculation, which lets the controller react quicker or slower, the derivative calculation dampens overshoot and oscillation around the target value.

The sum of these three calculations is then fed into the device we wish to control. Using the same example as earlier, this would be a cruise controller. The cruise controller then uses the newly acquired value to adjust the throttle and therefore the current speed, and the resulting speed is again fed back and lets us calculate a new error to use as an input.

### 3.2 Controlling a Web Cluster using a PID

To paraphrase [19]: a system level controller can make a cluster of low-power servers power proportional. The controller will use sleep states to switch CPUs on or off in order to continuously match the current workload with the system’s capacity. It uses methods from control theory to drive the CPUs from and into sleep states. Results from system simulation show that this approach can achieve a up to 80% reduction in energy consumption compared to a cluster using only DVFS.

In the previous section we used a cruise controller as an example to explain a PID controller. While applying the same concept to a web cluster’s power manager may seem different at first, it is actually fairly similar. We use the same formula - eq. 3.1 - to calculate our input, output and error values, they just refer to different things. Here we will use the system’s work load - in terms of number of requests per second - as our target value. System capacity is measured in maximum number of requests per second the system can handle, which, when we know the number of CPUs in the system, translates into the number of CPUs we need to meet the demand, or target value if you will. This is also our feedback value. The difference between the work load and number of active CPUs is then used to calculate the error value [19].

This scheme will not work if no one takes responsibility for turning CPUs on and off, which is why one core will be kept statically active. The static core will also be used to handle the low number of requests trickling in when the system is otherwise almost idle between peak hours. Currently, as the system in our reference was built around ARM Cortex-A8 CPUs, once a CPU is activated by the power manager (PM), it will run at the highest possible frequency. This is not, from a power consumption point of view, ideal, even though the ARM CPUs are designed to consume very little power [19][20].
Also, because the only scheduler currently available is the CFS, which by design uses all resources equally, we run into trouble whenever the workload drops. While the workload increases the PID will keep turning on CPUs to meet the demand, so they will all be fully loaded per definition. When the demand starts dropping off, however, CFS will start spreading tasks out over all the available CPUs. This in turn will mean the PID will be unable to switch the CPUs off. Therefore, as the PID is unable to influence scheduling, we will need a scheduler capable of reliably consolidating tasks on as few CPUs as possible, ideally with a negligible impact on performance. It is currently unclear as to whether or not CFS, even with _MC or _SMT, is capable of this. We will thus change the behaviour of the CFS to fill CPUs sequentially and its LB to always keep as many CPUs as possible fully loaded. These changes will then be compared against the original CFS both with and without its power-saving options turned on. This is how the work in this thesis relates to the PID Controlled Power Manager. We have been looking at ways to schedule tasks in a way that is beneficial both for the energy efficiency and performance.
In Chapter 3 we concluded an ideal scheduler for the PID PM would fill CPUs with tasks sequentially and its LB would ensure that we use as few CPUs as possible. When we herein say "sequentially" we mean; scheduling tasks onto one CPU until it has reached its maximum capacity. CFS does not currently do this - see Chapter 2. Even with its supposed power-saving options turned on, it will first schedule tasks onto all available CPUs and then during LB attempt to consolidate tasks. This is based on an impression gotten from reading the Kernel code however, so it will be tested in LinSched. LinSched as a tool is covered in Chapter 5.1.

In this chapter we will explain the changes made to the CFS’ code as an attempt to improve its power-saving capabilities. The scheduler we get from using these changes will henceforth be referred to as the Overlord Guided Scheduler (OGS). The name is a play at the fact that the PID, our Overlord, might decide to turn almost everything off at any time. Since we will only change the selection algorithms we will not have to concern ourselves with any locking mechanisms, which is quite fortunate since LinSched does not help verify locking. Still, to be on the safe side we will verify that changing the selection of RQs does not somehow circumvent the current locking mechanisms will. There was, however, no time to do this during this thesis project and it will thus be recommended as Further Work.

The changes suggested below, disregarding the proposed LB changes, have been spliced into the original CFS code at the appropriate places. The OGS code has been put behind \#ifdef declarations, and will thus only be included
and used when expressly chosen at the Kernel build stage. If we do not select OGS, then the CFS scheduler is used in its original state. Building a Kernel with the tweaked code and using only the default build options will result in the Kernel using CFS. We have attempted to splice the proposed scheduler changes into the existing scheduler instead of writing it from scratch because of the time constraints imposed on this thesis project. Had we decided to write one from scratch, we would, for instance, have had to understand and implement locking, which in itself would have easily used more than the amount of time we had available.

4.1 Runqueue Selection

Every new task has to be scheduled on a RQ on some CPU. For CFS this is always the idlest RQ, as was explained in Chapter 2.1.2. To make good use of the PID PM we want to always schedule tasks on the busiest non-overloaded RQ.

Using LinSched and a number of well placed text prints, as well as by looking at the Kernel code we identified `select_task_rq()` as the function making all the important decisions regarding RQ selection. This function then makes a task class-specific call to another function which makes the ultimate selection and returns it. We have not bothered with RT tasks due to the time constraints of this project, and thus the selection function we are interested in is `select_task_rq_fair()`. This function is called whenever a RQ is to be selected for a task falling under the influence of the CFS and we then proceed to make a RQ selection as was specified in Chapter 2.1.2. Looking at this function we can come to the conclusion that in order to change the RQ returned by this function, we need to change what `find_idlest_group()` and `find_idlest_cpu()` decide to return.

4.1.1 `find_idlest_group()` vs `find_busy_group()`

The current implementation of this function will, as the name implies, return the idlest SG from a given SD based on SGs’ average weighted load. The code below is copied from `find_idlest_group()` in the 2.6.35 version of the Linux Kernel. It is the part of that function upon which all decisions are based. The lines have been numbered after copying to make referencing it easier.

```c
1    /* Tally up the load of all CPUs in the group */
2    avg_load = 0;
```
for_each_cpu(i, sched_group_cpus(group)) {
    /* Bias balancing toward cpus of our domain */
    if (local_group)
        load = source_load(i, load_idx);
    else
        load = target_load(i, load_idx);
    avg_load += load;
}

/* Adjust by relative CPU power of the group */
avg_load = (avg_load * SCHED_LOAD_SCALE) / group->cpu_power;

if (local_group) {
    this_load = avg_load;
    this = group;
} else if (avg_load < min_load) {
    min_load = avg_load;
    idolest = group;
}

It is easy to see that the for-loop starting on Line 4 iterates through all members of the SG and tallies the total load. What is unclear is how the perceived load changes depending on whether the group is local or not - Lines 6–9. Based on the code comments we do know it is supposed to bias the selection towards a local group. Looking at the definitions of source_load() and target_load() in sched.c we can see that the bias lies in the size of the weighted_cpuload() that is added to avg_load on Line 11. This again takes us back to the problem of load weights and their lacking documentation. Sorting out the exact way the load weights work and are assigned was, as we mentioned earlier, deemed to exceed the scope of this project. Although the general impression is that assigning weights to tasks may not result in an ideal task distribution. This will affect our ability to detect the current load as well, what with the current upper limit also being based on load weights. The PID will need a feedback value telling us whether or not we are likely to need to turn on more CPUs, therefore we have another problem.

One solution to these problems would be to make these decisions based on how idle a CPU has been since we last scheduled a task on it. This is essentially already done by the Governors - covered in Chapter 2.2 - as they look at how idle a CPU is and adjusts the CPU’s frequency accordingly. As all the functionality is already in place, it would then be a short step to use
the same information when making decisions regarding RQ selection. For
the Governors to work, there needs to be a structure storing all the different
contributions to the CPUs utilisation. This structure is already defined in
`include/linux/kernel_stat.h`, so there is no need to define another. The update
functions for the CPUs’ busy times are also already in place. Using the
information gathered on behalf of the Governors to make our RQ selections
is then a trivial matter.

```
25 static struct sched_group *
26 find_busy_group(struct sched_domain *sd, struct
27     task_struct *p, int this_cpu,
28     cputime64_t time_now)
29 {
30     struct sched_group *busy_group = NULL,
31         *group = sd->groups;
32     // To keep track of the most loaded group
33     cputime64_t min_idle = time_now;
34     cputime64_t idle_time; // Time spent idle
35     cputime64_t SAFETY_MARGIN = 100000; //Arbitrary number
36     do {
37         int i;
38         /* Skip groups with no CPUs allowed */
39         if (!cpumask_intersects(sched_group_cpus(group),
40             &p->cpus_allowed))
41             continue;
42         /* Reset idle_time and safety for each group */
43         idle_time = 0;
44         safety = 0;
45         for_each_cpu(i, sched_group_cpus(group))
46         {
47             idle_time = cputime64_add(idle_time,
48                 what_is_idle_time(i, time_now));
49             safety += SAFETY_MARGIN;
50         }
51         if( (idle_time < min_idle) && (idle_time > safety) )
52         {
53             min_idle = idle_time;
54             busy_group = group;
55         }
```
Above we can see the code for the suggested replacement function without most of the comments, but otherwise in its entirety. When the new function is called, it takes essentially the same inputs as the old one, except we need to get the current time instead of load_idx. Another difference is that currently we do not get a specific SD to look in, as we have not yet changed the flags to be set appropriately for our purposes. Due to this we iterate through the available SDs looking for a suitable SG. To find a suitable SG we use the for-loop on Line 45, which sums up the total idle time in the group, as well as an, at this point, arbitrary safety margin. The function what_is_idle_time() was written specifically for the new RQ selection, and it’s functionality will be covered later in this chapter. When all the members of the SG have been accounted for, we move on to the conditional on Line 51, where we check if the current SG is more suitable than the SG currently perceived as the best candidate. An SG is perceived as a better candidate if:

- If the current SG’s idle time is smaller than the current best. During the first complete iteration, the current best idle time will be equal to the current system uptime. It is not possible for the SGs to have been idle longer than the system has been running. There also has to be at least one SG which has not been completely idle, since the OS has to run somewhere.

- If the idle time is greater than one safety margin per SG member, then the SG is deemed to not be overloaded. The safety margin is used because we currently have no idea about what the new task we want to schedule requires in terms of CPU time.

When this is done for all SDs and SGs available with the above conditions, we will end up with the SG which best matches the conditions at the time the function is called. Then, provided we have found a suitable
candidate SG, we return it to `select_task_rq_fair()` and used as an input in `find_busy_sched_cpu()`.

4.1.2 find_idlest_cpu() vs find_busy_sched_cpu()

The current implementation of this function suffers from the same problems as `find_idlest_group()`. As can be seen on Line 72 below, it is based around `weighted_cpuload()`. Using the SG reported as the idlest by `find_idlest_group()`, we iterate through the group members looking for the CPU with the smallest load. The found CPU is then returned to `select_task_rq_fair()` and will be selected as the target RQ for the new task. This, again, is not in line with our aims. We would like to select the RQ with the highest load available, while not overloading its CPU.

```c
70 /* Traverse only the allowed CPUs */
71 for_each_cpu_and(i, sched_group_cpus(group),
72    &p->cpus_allowed) {
73        load = weighted_cpuload(i);
74        
75            if (load < min_load || (load == min_load && i == this_cpu)) {
76                min_load = load;
77                idlest = i;
78            }
79    }
80 return idlest;
```

Below we have the suggested replacement function, which would work together with `find_busy_group()` in finding a suitable RQ to schedule on, in its entirety. Again without most of the comments.

The replacement function here is quite similar to the original in that it does the same kind of iteration - starting on Line 94 - through all the CPUs in the group. The main difference here is that, again, we look at how idle the CPU has been, instead of its `weighted_cpuload()`. Then, on Line 99 we get the RQ struct for the current CPU, which contains information about the number of tasks running on this CPU. We do this because we will, on Line 104, calculate how much execution time the currently running tasks require on average. This average will then be used as an estimate of how much execution time the new task is likely to require. Estimating task requirements using an average is not an ideal solution, as it is quite likely to over- or underestimate the amount of time the new task will require. Although, since we currently have no way of knowing
anything about new tasks, this solution will at least give us an estimate of how much busier another task might make the CPU.

```c
static int
find_busy_sched_cpu(struct sched_group *group,
                      struct task_struct *p, int this_cpu,
                      cputime64_t time_now)
{
    int busy_sched = -1;
    int i;
    cputime64_t idle_time = 0;
    cputime64_t avg_busy_time_per_task;
    cputime64_t min_idle = time_now;

    for_each_cpu(i, sched_group_cpus(group))
    {
        /* Get the time cpu(i) has spent idle */
        idle_time = what_is_idle_time(i, time_now);

        struct rq *arr_qu = cpu_rq(i);

        if (arr_qu->nr_running < 1)
            avg_busy_time_per_task = 0;
        else
            avg_busy_time_per_task = cputime64_sub(time_now,
                                                    idle_time)
                                      /(arr_qu->nr_running);

        if ( (idle_time < min_idle) && (idle_time >
                        (SAFETY_MARGIN+avg_busy_time_per_task)) )
            { min_idle = idle_time; //refresh the reference value
              busy_sched = i; //record which cpu this is
            }
    }

    return busy_sched;
}
```

Before calculating said average - on Line 102 - we make sure we have at least one task running before calculating the average. If no tasks are running on the CPU, then the average busy time per task is set to naught. Otherwise we
calculate the average normally. This is to avoid a potential division by zero. Once we have an average, we compare the idle time against the current best. We conclude we have found a new target CPU if:

- The current idle time is lower than the current best. The initial best value is, again, the current system uptime.
- The current idle time is higher than the sum of the safety margin and the average busy time per task. This is again a safety mechanism, which is used due to the uncertainty around the new task’s requirements.

Once we have found a suitable target CPU, we return it to `select_task_rq_fair()` and schedule the new task on it. If this function returns a `null` value, then we obviously need an alternative target CPU. This goes for the default CFS as well. Fortunately `select_task_rq()` - i.e. the function calling the task-specific RQ selection function from `sched.c` - already has a function taking care of this in `select_fallback_rq()`. This function has currently been left as is. Suffice to say it makes sure we find somewhere to schedule our task, even if our selection method found all CPUs to be unsuitable. At a later stage this function should also be looked at in order to find out whether or not it has to be changed.

The accuracy of our selection procedure will have to be evaluated, as it is quite heavily dependent on the average busy time calculation. If a CPU has a few tasks taking up a large portion of the available execution time, then the consequently high average may cause us to choose another CPU, even if the aforementioned CPU would be capable of handling a few low load tasks without being overloaded. It is possible that this problem is best solved during LB, and a possible, although not yet investigated solution will be mentioned in Section 4.2.

### 4.1.3 `what_is_idle_time()`

This function was put in `sched_fair.c` for the express purpose of getting and reporting a CPU’s idle time to OGS’ RQ selection functions. It is essentially identical to the equivalent function - `get_cpu_idle_time_jiffies()` - used by, e.g., the Ondemand Governor. The only differences being that the current system uptime here, `time_now`, is not a pointer and that we, on `Line132`, make sure we have no negative idle time values. It should not be possible to have a busy time that is longer than the system uptime, but due to some of the workarounds used in LinSched, detailed in Section 5.1.3, it did happen occasionally. The condition has been left in as a safety mechanism.
What this function then needs to work properly is a CPU, and the current system uptime. Which CPU we’ll be looking at is normally given by an unsigned int, which is basically the CPU’s ID number as far as the Kernel is concerned. The current system uptime is here stored in the variable \texttt{time\_now}. Its value is gotten from a check in \texttt{select\_task\_rq\_fair()} and given as an input to the \texttt{find\_busy\_\*} functions, which pass it on to this function. We are currently doing it this way as it keeps both the number of time checks down, and \texttt{time\_now} constant for the duration of the RQ selection process. There is a small chance that changing \texttt{time\_now} mid-selection will warp the result, hence its being constant while we are looking for a RQ.

Lines 127 – 130 are the ones where we actually add the busy times. Judging from the structure in \texttt{kernel\_stat.h}, the Kernel distinguishes between five different types of business, and these types are stored as elements in said structure. The total business is then the sum of the five business types, with the idle time - \texttt{idling} - being the difference between this sum - i.e. \texttt{busy\_time} - and \texttt{time\_now}. The difference is then returned to the calling function.

```
120 static inline cputime64_t
 what_is_idle_time(unsigned int cpu,
    cputime64_t time_now) {

121     /* Variables for tallying idle time */
122     cputime64_t busy_time; // Time spent busy
123     cputime64_t idling; // Temp variable
124
125     /* Tallying the idle time */
126     busy_time = cputime64_add(kstat_cpu(cpu).cpustat.user,
127         kstat_cpu(cpu).cpustat.system);
128     busy_time = cputime64_add(busy_time,
129         kstat_cpu(cpu).cpustat.softirq);
130     busy_time = cputime64_add(busy_time,
131         kstat_cpu(cpu).cpustat.steal);
132     busy_time = cputime64_add(busy_time,
133         kstat_cpu(cpu).cpustat.nice);
134
135     if (unlikely(busy_time > time_now))
136         idling = 0;
137     else
138         idling = cputime64_sub(time_now, busy_time);
139
140     return jiffies_to_usecs(idling);
```
It is possible we will not even need this function in a full Kernel adaptation, however the experimentation herein was done using LinSched, which only uses the parts necessary to emulate the normal scheduling subsystem, and no functionality from the \textit{CPU freq} subsystem was therefore available.

To reiterate then; the changes mentioned above, called OGS, will move us away from selecting the idlest RQ and to selecting the busiest RQ with some remaining capacity. The capacity is determined by summing up the different types of business stored in the \texttt{cpu\_usage\_stat} struct in \texttt{kernel\_stat.h}. We currently go through all the SDs to find the SG with the most suitable idle time, and then move on to find the CPU with the most suitable idle time in the previously detected SG. It is our feeling that making RQ selection decisions based on how idle a CPU has been may lead to a load distribution that is closer to the, for our purposes, ideal distribution compared to CFS’ scheme with weighted tasks. Said ideal distribution would have the tasks scheduled on as few CPUs as possible. It is also a lot easier to detect when a CPU is about to become fully loaded when using OGS, which is extremely important in order for the PID controller to be able to work properly, as we can simply look at how close to naught the idle time is. Neither do we need to store any information relating specifically to the capacity of the available CPUs, since, again, the information is already available in the idle time and its approaching naught. We therefore only need the current system uptime, the number of tasks belonging to the RQ, and how busy the CPU has been in order to make our RQ selection. All of these are already available in the kernel. The system uptime is stored in a variable called \texttt{jiffies}, the number of tasks in a RQ is an externally visible scheduling statistic, while the Governors keep track of a CPU’s busy time.

Figure 4.1 above illustrates how CFS and the weighted CPU load is assumed to affect the perceived business of a CPU compared to OGS and idle time. Each L-shaped block represents one task and its execution time requirements, while the space it takes up in the figure is equivalent to how the scheduler perceives it. Both schedulers have one CPU at their disposal in this illustration. Our assumption then is that the load weights may cause the CFS RQ selection to perceive each task as requiring an execution time equivalent to a 2-by-3 block instead of its actual requirement. Due to this, it might see a CPU as being fully loaded after 16 tasks have been scheduled on the CPU in question. The OGS on the other hand, is capable of detecting how much time the scheduled tasks have actually been using and is therefore able to schedule tasks in such a way that we use our CPUs more efficiently in terms of execution time. Again; this is
Figure 4.1: Illustration of how the weighted CPU load is assumed to affect perceived business

an assumption about the CFS and our aim for the OGS, and we will therefore attempt to validate said assumption and aim empirically.

The changes detailed in this section (4.1) only affect the RQ selection process. The function charged with finding a RQ for new tasks is the only one we have changed. Therefore, as CFS should work quite well for our purposes when using the OGS RQ selection, we use the original CFS code for, e.g., locking, and actually giving tasks CPU time.

4.2 Load Balancing

As was explained in Section 2.1.3, the CFS LB will by default attempt to achieve an even task distribution. This means the LB will always pull tasks from the busiest to this RQ. While CFS does have the SCHED_MC and SCHED_SMT options for pulling tasks from almost idle RQs, they only seem to work in cases where we have a as-good-as idle RQ, and pulling tasks from it would not upset the otherwise even task distribution, whereas we would like to minimise the number of CPUs in use by keeping the load as high as possible without overloading. We will attempt to show this in Chapter 6. Although we do have a relatively clear idea of what we would like to implement, we currently only have a superficial plan for affecting this change. The herein proposed changes have therefore not been tested.
The CFS LB works according to our needs as far as Step 5, by the reckoning used in Chapter 2.1.3. When we get as far as Step 5, we start looking for source and target RQs. We will refer to the RQ from which we pull a task as the source RQ, and to the RQ performing the pull as the target RQ here. At this stage CFS finds the busiest queue - find_busiest_queue() - in the busiest group - find_busiest_group(). When finding the group, CFS uses a struct called sd_lb_stats. This struct is updated every time the group-finding function is called, and contains information relevant to LB at this level. The elements are used at various points by different functions to achieve a balanced load.

### 4.2.1 find_busiest_group()

The elements used by find_busiest_group() to decide if we are imbalanced are:

- sds.this_load (struct sd_lb_stats sds) - Contains the perceived load of the we are a member of.
- sds.busiest - The currently busiest group
- sds.busiest_nr_running - The number of tasks running in the busiest group
- sds.max_load - The highest group load in this SD
- sds.avg_load - The average load in this SD
- balance - An integer value, used as a boolean, which indicates whether this is the appropriate CPU to perform LB at this SD level or not.

We previously - in Chapter 2.1.3 - mentioned that we check a number of cases which tell us if there is no imbalance from this CPU’s point of view if. These cases are:

- #1 This CPU is not the appropriate CPU to perform LB at this level. We check the value in balance and if it is 0 this CPU is not appropriate and we jump to ret, otherwise we move on.
- #2 There is no busy sibling group to pull from. Here we look at the busiest and busiest_nr_running elements in sds. If we either have no busiest group, or if there are no tasks running, i.e. busiest_nr_running == 0, we jump to out_balanced. In all other cases we move on.
• #3 This group is the busiest group. See if $sds\_this\_load \geq sds\_max\_load$, which if true means that this group is the busiest group. We want to pull from the busiest group, so we jump to $out\_balanced$.

• #4 This group is more busy than the average business in this SD. To be able to do this check then, we have to actually calculate an average load. This is done just before the actual if statement. If $sds\_this\_load \geq sds\_avg\_load$, then we do not want to pull tasks to this group, as it would not help us equalise the load.

• #5 The imbalance is within the specified limit. The check we do here is $100 \times sds\_max\_load \leq sd\_imbalance\_pct \times sds\_this\_load$. The value of $imbalance\_pct$ will depend on what SD we are currently looking at, but the idea is that its value will tell us how much larger $max\_load$ can be before we see it as being imbalanced. If the imbalance turns out to be withing the specified limits, we jump to $out\_balanced$. As an arbitrary example, let’s say $max\_load = 75$, $this\_load = 55$, and $imbalance\_pct = 125$. Using the formula above we would then get $7500 \leq 6875$, which tells us the imbalance is unacceptable. Although, were $this\_load$ to increase by 5 units, we would, using the same formula, get $7500 \leq 7500$, which is within the limits and prompts a jump.

• $ret$ - A point in this function we jump to from #1 when the CPU turns out to be inappropriate. Jumping here will return us to $load\_balance()$ with a null group, and with $imbalance = 0$ - the value of $imbalance$ will later let us know how much load to move in order to equalise the distribution.

• $out\_balanced$ - A point in the function we jump to if one of the checks #2-5 tells us there was no obvious imbalance. $check\_power\_save\_busiest()$ is called from here to see if we have opted for power-saving and whether or not it is possible to perform such a balancing. Whatever value $check\_power\_save\_busiest()$ returns is then returned to $load\_balance()$.

If we clear all these checks, then we will calculate how imbalanced we are using $calculate\_imbalance()$. The value returned from this function is the amount of weighted load we have to move to equalise the distribution from the point of view of $this\_cpu$. We will cover this function, as well as how we would like to change it, in Subsection 4.2.3 later in this chapter.

As was the case with RQ selection, CFS works counter to our purposes. Now, again, we did not manage to implement LB for OGS within the timeframe of this thesis, so the LB changes proposed herein have not been tested and are only a draft design. We have yet to identify all scenarios where an imbalance does not exist from the point of view of $this\_cpu$. One scenario we will most
Certainly need to check is if this_cpu is the idlest one. If it turns out to be the idlest, then we want to pull tasks from it and not the other way around. Another scenario would be that this_cpu has no idle time to spare, or its excess idle time is smaller than the average busy time per task of idlest. Other scenarios are quite likely to exist, but have yet to be identified.

It is our feeling at this point that an equivalent of the check_power_save_busiest() is not needed, since we are attempting to save power by default when using OGS. A possible substitute for this function could then be a check to see if one of the busier RQs are overloaded, and whether or not this_cpu could alleviate the problem. However, this is currently not possible, as we have not implemented a way of detecting overloads. We can easily detect when the idle time is approaching naught, but if the busy time were to exceed the available execution time - exceeding the execution time should here be seen as a problem akin to RTS, where all tasks do not fit in a certain period, and thus we start missing deadlines - as the idle time can never be negative. This could potentially be solved by borrowing from RTS theory. Each task would upon creation be given a soft deadline, or an acceptable waiting time if you will. Once tasks start to occasionally miss these deadlines we can assume that the RQ is on the verge of becoming overloaded. Once they reliably miss their deadlines, the RQ is definitely overloaded. This solution is, again, only a hypothesis and has neither been implemented nor tested.

4.2.2 find_busiest_queue()

When looking for the busiest queue we do not need to use an SG struct in the same way find_busiest_group() used an SD struct. The search has now been refined sufficiently for it to be a lot more efficient to look directly at the RQs of all the CPUs in the chosen group. Using the loop for(i, sched_group_cpus(group)), Line 1 below, we make sure that we iterate over all the SG members. The unsigned long power will contain what is supposed to be a measurement of the capacity of the current CPU, and the unsigned long capacity is then the measured capacity scaled by a constant; SCHED_LOAD_SCALE. The unsigned long wl will contain the detected workload for the CPU we are currently looking at, which we get using a call to weighted_cpuload(). We also get the RQ struct for the current CPU. The RQ that is returned after the last iteration will be the one with the highest wl value - wl is scaled by CPU power, on Line 12, before being compared against the other loads. The if statement on Line 13 then compares the current CPU’s load against the currently highest load, and stores the new values if they turn out to be higher.
for_each_cpu(i, sched_group_cpus(group)) {
    unsigned long power = power_of(i);
    unsigned long capacity = DIV_ROUND_CLOSEST(power, SCHED_LOAD_SCALE);
    unsigned long wl;

    if (!cpumask_test_cpu(i, cpus))
        continue;

    rq = cpu_rq(i);
    wl = weighted_cpuload(i);

    /*
     * When comparing with imbalance, use weighted_cpuload()
     * which is not scaled with the cpu power.
     */
    if (capacity && rq->nr_running == 1 && wl > imbalance)
        continue;

    /*
     * For the load comparisons with the other cpu’s, consider
     * the weighted_cpuload() scaled with the cpu power, so that
     * the load can be moved away from the cpu that is potentially
     * running at a lower capacity.
     */
    wl = (wl * SCHED_LOAD_SCALE) / power;

    if (wl > max_load) {
        max_load = wl;
        busiest = rq;
    }
}

In order for the OGS LB to work as planned, we would like to find the idlest queue with at least one task running. One possible solution would be along the lines of the code below. Here we would iterate through the SG members using the same for-loop as with CFS. Now though we will start out by declaring an unsigned long \(it\), short for idle time, which we will use to find the largest idle time of any CPU and RQ with at least one task running. The if statement on Line 26 will force us to move on to the next iteration if the current RQ has no tasks running, as per the idlest queue with at least one task running requirement, and the if statement on Line 28 will store the RQ with the largest idle time. The RQ will still be stored in busiest, which may sound like a curious
choice for a variable’s name, but since OGS has essentially been spliced into the original CFS code, we have tried to change as little as possible to ensure both schedulers can be compiled interchangeably from the same code.

```c
20 for_each_cpu(i, sched_group_cpus(group)) {
21 unsigned long it;
22 if (!cpumask_test_cpu(i, cpus))
23 continue;
24 rq = cpu_rq(i);
25 it = what_is_idle_time(i, time_now);

/* **
 * If rq is empty, then skip it.
 * There should never be a case where ALL rqs are empty,
 * since LB is invoked AFTER we’ve got a few tasks running
 * nr_running < 1, since we don’t want to pull from an empty RQ */
26 if (rq->nr_running < 1 )
27 continue;

/*
 * For the load comparisons with the other cpu’s,
 */
28 if (it > max_idle) {
29 max_load = it;
30 busiest = rq;
}
}
```

Again; these changes have not been tested yet, but, provided the group-finding function works as intended, they should work more or less as-is. The testing of these functions subsequently has to fall under Future Work.

Another couple of things that will have to be worked on in the future are the functions `update_sd_lb_stats()` and `update_sg_lb_stats()`. These functions update their respective LB stats structures, and currently only do it for the benefit of CFS. If the OGS LB is to work properly, we need to either tweak these functions or write new ones which will update an SD- and an SG-related struct with data relevant for OGS LB.
4.2.3 calculate_imbalance()

To paraphrase the comments in the code: calculate_imbalance calculates the amount of imbalance within the groups of a given SD during LB. As inputs it uses sds, which was earlier defined as SD statistics relevant to LB, this_cpu, which is the CPU for which LB is being performed - and in the case of CFS LB, the CPU to which tasks will be pulled - and then output the results to the variable imbalance. All of the code excerpts here are from the function calculate_imbalance() in sched_fair.c.

```c
sds->busiest_load_per_task /= sds->busiest_nr_running;
if (sds->group_imb) {
  sds->busiest_load_per_task =
  min(sds->busiest_load_per_task, sds->avg_load);
}
```

Before CFS calculates the actual imbalance we check if an SG has reported an internal imbalance. An SG considers itself imbalanced if the difference between its max_load and min_load is more than twice the avg_load_per_task in the group. This is checked and potentially updated every time update_sg_lb_stats() is called, which happens as an extension of the initialisation calls at the start of find_busiest_group(). If an SG has reported being internally imbalanced, CFS will use the smaller value of busiest_load_per_task and avg_load as its recorded busiest_load_per_task value, as can be seen from the code excerpt above.

According to the comments in the Kernel code, there are some cases when using SMP Nice balancing where max_load is smaller than avg_load. Judging from the comment, SMP Nice will skip groups at or below its capacity while calculating max_load [3]. Whenever this happens, calculate_imbalance will return an imbalance of naught and call fix_small_imbalance() before returning to find_busiest_group().

Then, if we did not find one group to be internally imbalanced, CFS will take a look at the busiest SG to see how much more than its capacity it is currently executing. This check is performed by the code excerpt below, where load_above_capacity will be scaled according to the constant SCHED_LOADSCALE and the SG’s scaled capacity. The resulting value should be something along the lines of a weighted percentage above capacity.

```c
if (!sds->group_imb) {
  /*
   * ...
   */
```
* Don’t want to pull so many tasks that a group would go idle.
*/
load_above_capacity = (sds->busiest_nr_running - sds->busiest_group_capacity);

load_above_capacity *= (SCHED_LOAD_SCALE * SCHED_LOAD_SCALE);
load_above_capacity /= sds->busiest->cpu_power;
}

Next, CFS will look for the minimum possible imbalance to correct. Judging from the comment left at this stage of the code, the reasoning behind this is that if all SGs take on a small amount of the load perceived as imbalance, then the distribution will eventually be equalised. By this reckoning then, CFS will report the smallest detected imbalance, as per Line 1 in the excerpt below. On Line 2 CFS then checks how much of the imbalanced load this _cpu could pull. If the scaled difference between the average load and _this_load is greater than the amount of detected imbalance, then the function will report that all of it can be pulled.

/*
 * We’re trying to get all the cpus to the average_load,
 * so we don’t want to push ourselves above the average load,
 * nor do we wish to reduce the max loaded cpu below the
 * average load. At the same time, * we also don’t want to
 * reduce the group load below the group capacity (so that we
 * can implement power-savings policies etc). Thus we look
 * for the minimum possible imbalance.
 * Be careful of negative numbers as they’ll appear as very
 * large values with unsigned longs.
 */
1 max_pull = min(sds->max_load - sds->avg_load, load_above_capacity);

/* How much load to actually move to equalise the imbalance */
2 *imbalance = min(max_pull * sds->busiest->cpu_power, (sds->avg_load - sds->this->cpu_power, (sds->avg_load - sds->this_load) * sds->this->cpu_power)/ SCHED_LOAD_SCALE;

On the off chance the detected imbalance is negligible, there is no guarantee anything will be moved so before returning, CFS will do one last check to see whether or not _imbalance is larger than the average load per task on busiest
- i.e. busiest_load_per_task - which is the future pull target. If this turns out to be the case, then CFS will call fix_small_imbalance() to ensure the LB takes a closer look at load distribution, even though the detected imbalance is negligible. Although, there is no guarantee fix_small_imbalance() will actually move anything.

In our case, i.e. the OGS imbalance calculation, we would like to know how much below capacity the CPUs in the busiest groups are. As we have said before, the purpose of OGS is to use as few CPUs as possible by filling them to capacity. An imbalance in our case then would be when this_cpu has an idle time larger than the average busy time per task and is not ranked as idlest. The planned LB procedure would then look at the average load per task in the idlest RQ, with at least one task running, compare this value against the average load per task in this_cpu, and pull as many tasks as this comparison says is possible. The OGS imbalance calculation would then be something along the lines of:

$$imbalance = \frac{this_cpu.idle_time}{idlest.avg_busy_time_per_task}$$ (4.1)

where imbalance - this value will have to be rounded down and compared against the number of tasks running on idlest ensure we do not attempt to pull more tasks than are available - is the number of tasks we want to pull, this_cpu.idle_time is the current excess capacity on this_cpu, and idlest.avg_busy_time_per_task is the average contribution per task to the business of idlest. Calculating the imbalance in this fashion should take care of safeguarding against pulling too many tasks to this_cpu while still pulling as many tasks as possible from idlest. Of course, this is still only, as has been mentioned several times with regards to OGS LB, a hypothesis. A heads up of how we would like to change it if you will, so actually implementing and testing this will fall under future work.

4.2.4 move_tasks()

After we have found both a group and a RQ it is time to move tasks and attempt to equalise the distribution. As its name implies, this is done using the function move_tasks(). The call returns the amount of load that was actually moved, which if greater than naught indicates the LB was successful.

The moving procedure consists of calling load_balance_fair() repeatedly while the moved load is smaller than the reported imbalance. This function does not seem to need any changes. If we have managed to change the LB procedure
successfully, then all the variables necessary to move tasks according to our wishes should have been included in the function call. Then, from this function on it seems we need not change anything.
5.1 Linsched

Scheduling is one of the most complex aspects of the Linux kernel. Developing schedulers or changing policies and thereafter verifying their behaviour can be very challenging without the necessary experience. Fortunately there have been some recent advances in this area with the development of LinSched. LinSched is a user-space Linux scheduler simulator which hosts the prototype scheduler in user-space and models it against arbitrary hardware targets in order to validate said prototype across a wide spectrum of topologies [15].

Traditionally new schedulers and policies would have had to be implemented directly in the kernel, which, due to the depth at which the scheduler resides, made measuring performance and debugging difficult. Adding traces could potentially change the scheduler’s behaviour and possibly even hide defects and inefficiencies. Not to mention that repeating the procedure for every topology one wishes to use is too time-consuming to be practical. LinSched circumvents these difficulties by migrating the scheduler sub-system into user-space and adds a thin emulation of the components necessary for the scheduler to be able to execute outside of the kernel. In order to be able to provide the stimuli necessary for workload validation it also builds a set of application programming interfaces (APIs) around it. This also allows us to gather enough relevant data to understand and measure the scheduler’s behaviour. Another upside to this approach is that a misbehaving scheduler potentially causing a
5.1.1 LinSched Tasks

A task in LinSched can be one of four types; normal, batch, first in-first out (FIFO) or round robin (RR). Their sleep and run times can also be specified, but will be static, insofar the scheduler actually serves them, once the simulation is running. They are also currently persistent in the sense that they will want to run periodically for the duration of the simulation. It is also possible to define a subset of CPUs on which a task is allowed to run. When using CFS this does not affect the RQ selection in any way other than forcing the tasks to run on the
CPU, or CPUs, to which they are bound. When using OGS on the other hand, we should see the CPU, or CPUs, to which tasks have been bound being filled up first. CPU binding will thus be used to ensure that OGS selects the correct RQ to fill up first [15].

5.1.2 Simulator Limitations

The task persistence is, for our purposes, a quite severe limitation as it will affect the LB and making it essentially impossible to tell if the LB works as intended. It is possible to ensure we get, for the scheduling policy and topology in use, an ideal task distribution, but since there is no way of removing tasks from the simulation once they have been created, we have no way of knowing how the LB will behave when the scheduling circumstances change. This has to be taken into consideration when interpreting the results gathered from LinSched.

Another limitation in LinSched is caused by the emulated kernel not being a full version. Due to this we are missing some functionality needed for OGS to work properly. Since OGS bases its scheduling decisions on the busy and idle time of individual CPUs, there is a need for timing updates in order to keep track of the amount of time each of the CPUs have been busy. These timings need to be stored in a CPU-specific structure as well. There is also a need for a variable keeping track of the amount of time the simulation has been running in order to be able to calculate for how long a CPU has been busy. None of these features are included in LinSched, which necessitates a workaround.

5.1.3 Workarounds

The time-keeping issue was solved by using the jiffies value from the HOS and adjusting that using a modulo operation where the divisor equals the simulation time in milliseconds. This is not a perfect solution, as there is no guarantee the time will start from naught nor that it will not overflow mid-simulation. It is however a solution that is good enough to make sure OGS works as intended.

To save the CPU’s busy times the kernel normally uses a structure defined in a header file `kernel_stat.h`. Normally there would be one structure defined per CPU, but this is not the case in LinSched. Most likely because the scheduler code is only compiled once and then just in a generic sense and not based on the topologies we intend to simulate. To get around this we have opted to use an identical structure in a separate file called `faked_kstat.h` and define an
array of 100 samples of the structure therein. As the largest number of CPUs currently simulated is 24 we thus have more than enough for our purposes. It was decided that 100 structures was a suitable number in case there was ever a need to simulate a larger system and if need be this number can be changed to whatever size we wish simply by changing the value of a constant defined at the top of the file containing the relevant code; \textit{sched\_fair.c}.

Linux’s default timing update functions are not included either, essentially meaning that there is no busy time accounting in LinSched. This is crucial for OGS and the scheduling decisions it makes. To solve this we have added functions which fake the aforementioned busy time accounting. They are quite simple and just add an arbitrary time to each of the elements in the time-keeping structure. As we want the updates to resemble an actual system the update function calls have been added to the functions in \textit{sched.c} which are called when a task is created - \textit{wake\_up\_new\_task} - and when a task is woken up to be put on a CPU - \textit{try\_to\_wake\_up}. While this solution means the recorded runtime does not necessarily correspond to the actual one defined in the LinSched simulation script, it does mean that we can adjust the amount of busy time a task contributes and since OGS bases its decisions on these figures it will still let us verify and validate the behaviour of the RQ selection algorithm.

5.2 Simulated Workloads

As the purpose of OGS is to ensure a CPU’s time is used as efficiently as possible, we want to test various different loads where the CFS displays a less than ideal time usage. In order to get a more complete picture of how the two schedulers compare, we will also run tests with workloads where the CFS performs well by definition. The different cases will be outlined here and the specific parameters used will be covered in Chapter 6 where the acquired results are analysed.

One case for which the CFS’ performance, for our purposes, is less than ideal is when there are a large number of tasks that require a low amount of run time. Using the default settings for CFS these tasks will be spread out evenly over all available CPU RQs regardless of how much processor time they actually need. Using all CPU’s in a case when all tasks could be run without, or with a negligible, overhead will dissipate more energy than is necessary. The \textit{CONFIG\_SCHED\_MC} option is supposed to accomplish something similar to OGS, but since it is only an alternative LB algorithm there is no guarantee it will not start by using more CPU’s than is necessary. Neither is there - short of
dissecting the inner workings of the task weight calculations, which could not be done due to the time constraints - any way of defining an accurate upper limit to the number of tasks that can be scheduled on a CPU before it should be considered fully loaded.

The OGS on the other hand bases its decisions on the amount of time a CPU has been idle since we last scheduled something on it. If the idle time is deemed large enough we schedule another task on that CPU’s RQ. When the tasks require very little time on a CPU most of them should fit onto a just a few RQs, meaning that if the load stays more or less constant the CPUs that are not in use can be turned off. However a risk with using OGS is that a CPU may become overloaded. It may be due to a large number of new tasks arriving at the same time, in which case the tasks may not have time to leave an accurate footprint on the busy time. Another potential risk is the LB. If it does not behave as intended it could pull tasks onto the wrong - the wrong RQ here being either an overloaded or a barely loaded RQ - CPU RQ which would lead to a suboptimal task distribution. Neither is there currently an OGS specific fallback algorithm for cases when no suitable CPU is found for new tasks, meaning we utilise the same one as for CFS.

Our simulated workloads should thus attempt to highlight the schedulers’ behaviour under different circumstances. We would like to know what kind of threshold CFS, with the `CONFIG_SCHED_MC` option turned on, has when deciding whether or not to use a new CPU and RQ, and if its performance suffers in any way compared to the default CFS. We would also like to find out if OGS behaves as intended and whether or not it could potentially save us power without an unbearable decrease in performance.

LinSched writes the results from these simulations straight to the screen, which makes reading the results from simulations with a large number of tasks quite tricky. We therefore piped the output to a text-file, and then copied the parts of the output we needed to a Spreadsheet. The figures in Chapter 6 were then drawn therein. The result comparisons we are interested in at this point are:

- Execution, and Run Delay times per task. To tell whether or not the schedulers’ performances are comparable
- Which CPU a task is allocated to, and in which order.
- The number of tasks on each CPU at the end of the simulation. To estimate power consumption per scheduler
- Ratioed Execution times. To tell how large the difference between execution times are in relative terms. An execution time difference of 1000 ms, while still significant, is a lot less dramatic if the tasks ran for a
total of, say, 1 000 000 ms, than had they run for 10 000 ms. To do this we will divide the CFS Execution times by their OGS counterparts, as per:

\[
Ratio = \frac{CFS(task_i(exec\_time))}{OGS(task_i(exec\_time))}
\]  

(5.1)

where \( task_i(exec\_time) \) is the Execution time of the task with ID \( i \). This will then mean that a \( Ratio < 1 \) points to OGS performing better by a factor of \( \frac{1}{Ratio} \), while a \( Ratio > 1 \) indicates CFS performed better by a factor of \( Ratio \).

### 5.3 Power Consumption

While we have not yet been able to measure power consumption in an actual implementation of OGS, we can still get a rough idea of its performance by using a constant per-CPU power consumption value. This value will be chosen almost arbitrarily, and lean towards a value that makes calculation easy. We will assume we are not using Governors, and that the power consumption of each unused CPU is instantly naught. Essentially we will assume the ”horse” is a ”sphere” to make the math easier [4].

We will assume each used CPU results in a 10W power consumption.
RESULTS

Here we will cover the results gathered using the methodology explained in Chapter 5. The amount of results produced is too big to cover all of them in this thesis, so we will only cover a subset. This subset will be chosen in a manner which is meant to illustrate both cases where the OGS performed well, as well as cases where it performed poorly.

The following figures - Figures 6.1, 6.2, 6.3, and 6.4 - compare a LinSched run for CFS and OGS with the following parameters:

- Topology: Dual-Socket Quad CPU
- Simulation time: 120 000 ms (ticks) = 2 minutes
- CPU-bound tasks: 2, bound to CPU 1
  - Sleep Time: 50 ticks
  - Run Time: 300 ticks
- Other tasks: 10, runnable on ALL
  - Sleep Time: 2000 ticks
  - Run Time: 10 ticks
- LinSched jiffies = $jiffies \% 200000$
- Safety Margin = 1000
• Faked busy time per task = 4 * 250

The simulation is run for 1000 ticks when a CPU-bound task is created, as these had a longer Run Time, and for 50 ticks when other tasks are created. The faked busy time is added to the struct in fake_kstat.h whenever `wake_up_new_task()` or `try_to_wake_up()` are called.

![Graph](image1)

Figure 6.1: Execution and delay times for a Dual Socket Quad CPU

Figure 6.1 above shows us the execution and delay times for all tasks and both schedulers. As we could have guessed from the parameter set, the CPU-bound tasks, i.e. the first two tasks in the figure, have much longer Run and Delay times than the other tasks. We can also see that CFS lets these two tasks run for longer than OGS. Otherwise both schedulers give all our tasks a similar amount of execution time. The bumps in the graph from Tasks 8 to 10 are LinSched bugs of some sort, as these tasks ran for longer than they, according to the parameter set, should be able to. There were a number of similar occurrences in a few other simulations as well, and the spikes they cause should always be disregarded unless we have stated otherwise.
Here, in Figure 6.2, we can see the relative difference in execution times. Each CFS task has been divided by its OGS counterpart, i.e. the OGS task with the same task ID, to make it easier to tell how large the difference is in relative terms. We can then see, in Figure 6.2, that the performances of the two schedulers were, apart from the aforementioned bugged values for tasks 8, 9, and 10, more or less identical. The two CPU-bound tasks, 1 and 2, do show that CFS performed better, but this performance is only better by a factor less than 1.05. This is in our opinion negligible, especially when we look at how the tasks were distributed.

Figure 6.3: Order of CPU allocation for a Dual Socket Quad CPU
Figure 6.3 then shows which tasks were sent to which CPU and in which order. We can - at least for OGS which currently has no LB and thus has no way of rearranging the task distribution during a simulation run - therefore tell in which order the schedulers fill the available CPUs. Here we can see that OGS only uses a single CPU for all the tasks, while CFS uses two and in separate sockets to boot. The CPU IDs in all Allocation Figures should be adjusted by \(-1\). This is because Excel was obstinate and would not let us start with \(ID\ 0\). Considering results seen in the two previous two Figures, i.e. 6.1 and 6.2, using more than one CPU is not necessary. This indicates that OGS can indeed improve performance from a power efficiency point of view.

Figure 6.4 shows the number of tasks per available CPU. This is essentially a snapshot of the distribution at the end of the simulation. Here we are therefore not concerned with the temporal aspect present in Figure 6.3, which is temporal in the sense that tasks are created sequentially and in-order and the simulation is then run for a pre-defined number of ticks before creating the next task, or, depending on the simulation setup, batch of tasks. In any case, Figure 6.4 confirms that OGS only uses one CPU compared to CFS’ two CPUs. This would, based on the Power Consumption estimation explained in Section 5.3, equate to OGS using half as much power as CFS.

The next set of Figures - 6.5, 6.6, 6.7, 6.8 - were produce using almost the
same set of parameters as the previous set. The only difference being the total number of tasks and the CPU topology. Here we have only 8 tasks, of which 2 are bound to CPU 1 with a higher CPU usage requirement, and are using a Multi-core Quad CPU topology.

In Figure 6.5 we can again see that the execution times of both schedulers are roughly the same. OGS does give the two CPU-bound tasks less execution time, but the difference is quite small. The other tasks get essentially the same execution times regardless of the scheduler.

![OGS vs CFS - Quad CPU Multi-Core](image)

Figure 6.5: Execution and Delay Times per task for a Multi-core Quad CPU

Figure 6.6 shows what we implied in the previous paragraph: the relative differences in execution time are small enough to be negligible. Most of the tasks favour CFS by a factor less than 1.005 and even the largest difference, namely tasks 1 and 2, favour CFS by a factor of roughly 1.023. Then when we look at Figure 6.7 we can again see that OGS has favoured using only one CPU, where CFS uses two.
Figure 6.6: Execution Times ratioed per task for a Multi-core Quad CPU

Figure 6.7: Order of CPU Allocation for a Multi-core Quad CPU

Figure 6.8 again confirms the number of CPUs used by each scheduler, and, again, OGS uses half the power CFS does.

However, these results are not conclusive. First off they only represent a subset of the simulated topologies. Secondly, these results also represent cases where OGS worked as intended. As we will now demonstrate, this was not always the case.

The next set of results were produced using the following parameters:
Figure 6.8: Tasks per CPU for a Multi-core Quad CPU

- Topology: Dual-Socket Hex CPU with SMT
- Simulation time: 120 000 ms (ticks) = 2 minutes
- CPU-bound tasks: 2, bound to CPU 1
  - Sleep Time: 50
  - Run Time: 300
- Other tasks: 260
  - Sleep Time: 2000
  - Run Time: 10
- LinSched jiffies = jiffies\%120000
- Safety Margin = 100 000 000
- Faked busy time per task = 125

While the topology here implies the use of SMT, it will only, as far as LinSched and the simulation is concerned, result in the double of the amount of available CPUs and not in any actual use SMT. At least not as far as we have been able to tell. The total number of CPUs here is then: 2 * 6 * 2 = 24.
We will again start by looking at the execution and delay times, i.e. Figure 6.9. The first thing we notice here is that the delay times for OGS are on average a lot higher than for CFS, though the execution times seem to be unaffected. This can most likely be chalked down to the very low run time required for the unbound tasks. They may have to wait a long time for their portion of the execution time, but once they do get it they finish quite quickly and the effect of a delay is relatively small.

Figure 6.9: Execution and Delay Times per task for a Hex CPU with SMT

Figure 6.10 shows that the execution times were not completely unaffected by the delays. CFS manages to run the unbound tasks for roughly half again as long as OGS. The spikes in the figure are, again, some form of bug where the tasks have exceeded their total possible execution time.

Figure 6.11 is a prime example of what we are trying to avoid using OGS. Here we can clearly see how CFS spreads its tasks out over all the available resources, which results in a sawtoothed waveform. On the other hand this figure is also a good example of what we get when an OGS simulation fails. The two CPU-bound tasks are correctly scheduled on CPU 1, but all the other tasks are scheduled on CPU 0. We have come to the conclusion that this occurrence is related to the conditions surrounding the choice of CPU, and the potential subsequent choice of a fallback. Whenever OGS fails to find a suitable CPU onto which to schedule new tasks, CPU 0 will be chosen as a fallback. In this particular simulation it happens quite often, as the large size of the safety margin combined with the jiffies value used by LinSched to make scheduling decisions being prone to overflow raises the probability by a considerable factor. Using the current testing method there is no way to
ensure against an erroneous LinSched jiffies, because the simulator itself does not keep count and runs according to its own time, which is quite a lot faster than the simulated two minutes, to boot. An overflow of the LinSched jiffies will then result in the perceived idle time likely being naught for the rest of the simulation, especially when combined with a large safety margin like the one used here. A better solution could have been used, but we thought it quite fitting to compare CFS against a scheduler prone to problems like the aforementioned. This, however, only makes sense from a test perspective, where one wishes to find out what the worst-case scenario for the proposed scheduler looks like compared to an average run for the de facto standard scheduler.

In our opinion, the apparent failure of OGS only serves to increase impressiveness of the numbers in Figure 6.10. That CFS only manages to serve the tasks for half again as long, even though OGS, as can be seen in Figure 6.12, used only two CPUs - one for the CPU-bound tasks, and one for the remaining 260 - makes for a compelling argument for the possibility of improving on CFS’ energy efficiency. However it should be kept in mind that the tasks used for these results required very little CPU time each. Failing to schedule tasks onto a CPU other than CPU 0 when said tasks require more time to execute would not have yielded similar results. This problem has to be solved before OGS can be safely used, and will thus be listed under Further Work.
If we again base our power consumption estimate on the conditions specified in 5.3, and use the task distribution and CPU usage seen in Figure 6.12; we can see that CFS uses a whopping 240 W, while OGS only uses 20 W. This is
quite a significant difference, especially considering the comparatively small
difference in execution times - as seen in Figures 6.9 and 6.10. It also introduces
what is likely to become the crux in the choice of schedulers. Either we
prioritise performance (CFS) or we prioritise energy efficiency (OGS).

The following results can not really be used in a comparison of the two
schedulers, as we, for some reason, seem to have neglected to save the
parameter set for this particular simulation. They are only included because
they, to a degree, illustrate the behaviour we wish to see in a scheduler that
could be used in conjunction with the PID PM. However, we will still take a
quick look at the CPU allocation and use, as well as the execution time ratios
to show an indication of how the aforementioned ideal attributes would affect
a system.

![OGS vs CFS - Quad CPU Quad Socket](image)

**Figure 6.13: Order of Allocation for a Quad Socket Quad CPU**

Figure 6.13 show the order of allocation for a Quad Socket Quad CPU system
under the circumstances mentioned in the previous paragraph. During the
allocation of the first 256 OGS tasks here, we can see a tendency towards the
square wave-like pattern we wish to accomplish. This pattern should, as tasks
are created with sequential IDs and OGS currently uses no LB, indicate that
CPUs are filled to the brim before tasks are allocated to an empty RQ. Although
after the first 256 tasks there is some kind of breakdown in the algorithms.
From this point onward OGS only allocates tasks to **CPU 0**. Attributing this to
something specific is difficult without the simulation parameters, but as **CPU 0**
seemed to be the fallback CPU of choice in our LinSched simulations it is likely
due to a jiffies overflow, a fault in the faked busy time update, or a similar issue
which would lead to OGS not being able find a suitable target RQ. CFS on the
other hand clearly displays a behaviour we would, ideally, like to avoid in that
it essentially allocates tasks to the CPUs one by one. This is especially evident
in the mid-part of Figure 6.13.

![Execution Times Ratioed](image)

**Figure 6.14: Execution Time Ratios for a Quad Socket Quad CPU**

As could be expected, and is confirmed in Figure 6.14, OGS tasks allocated
to CPU 0 were given atrociously low execution times compared to their CFS
counterparts. The spike in the shape of a cathedral in the first fifth of the figure
is also caused by tasks allocated to CPU 0. All other OGS tasks got as long, or
longer execution times than their OGS counterparts.

Figure 6.15 again illustrates the number of CPUs used by either scheduler.
Were not OGS’ usage of CPU 1 - i.e. what due to an annoyance in Excel reads
as CPU 2 in the figure - disappearingly small in this context, we would see that
OGS uses only four CPUs, while CFS uses all 16 CPUs. This would likely not
have been the case had OGS yielded results as intended and the 1600 tasks on
CPU 0 had been spread out more evenly. However, this would, in a kind of
worst case scenario, only equate to OGS yielding a task distribution similar to
CFS’, but with a different order of allocation.

The conclusion one can draw from the results presented in this chapter is
that OGS could indeed provide some energy-efficiency improvements. Even though the current OGS implementation is bugged and lacking any form of LB, we could see that it is possible to use fewer CPUs with a negligible performance sacrifice compared to CFS. On thing that should be kept in mind is that the simulations, on which the figures are based, did not use real tasks. The LinSched tasks will, once created, exist, and run periodically, for the duration of the simulation. Neither were these simulations really based on systems in which OGS is likely to be used - that is to say; a system where a PID Controller turns CPUs on and off depending on the current load - so we do not yet know how a sudden load burst which exceeds the current system capacity will affect performance. This is also, after an LB has been developed and implemented, the next logical step; simulate the scheduler using a full kernel and a tool which is closer to a real system, e.g. Simics. Simics could possibly let us test whether or not using a Governor in conjunction with OGS is possible with the current implementation. There is a chance that the Governor’s adjusting the CPU frequency to match the current load will lead to OGS perceiving all CPUs as fully loaded. If this turns out to be the case then we would, for instance, need to devise a way of detecting the current CPU frequency of an arbitrary CPU and, while the frequency is less than the allowed maximum, base decisions on this value instead of the idle time. In short then: OGS has a niche it can fill, was not finished during this thesis project, and needs to be further developed before it can be safely used in any real-life system.
FURTHER WORK

This chapter will list OGS issues and problems that have been identified, but have yet to be solved. It will also list what is not so much issues with the scheduler, but rather functionality that has yet to be implemented.

• Test in a full-kernel simulator. E.g. Simics.

• Get a concrete measurement of power consumption when using OGS, and OGS in its intended role as a part of the PID PM.

• Implement and test OGS LB as proposed
  -- Add struct elements relevant to OGS LB to struct sd_lb_stats. Put them behind #ifdefs with conditions ensuring they are only built in and used when OGS is the scheduler of choice.
  -- Add struct elements relevant to OGS LB to struct sg_lb_stats. Put these elements behind #ifdefs as well.
  -- Change update_sd_lb_stats() in order to update the struct elements relevant to OGS LB.
  -- Change update_sg_lb_stats() in order to update the struct elements relevant to OGS LB.
  -- Develop alternative to find_busiest_group(). Call it find_idlest_group() or similar.
  -- Identify scenarios for OGS where an imbalance does not exist from the point of view of this_cpu.
– Change `find_busiest_queue()` to `find_idlest_non_idle()` or similar.

– Change the calculations in `calculate_imbalance()` to the ones suggested in Subsection 4.2.3.

– Verify that everything works as intended, without changes, from `move_tasks()` all the way to the end of the LB procedure chain.

– Solve the fallback problem. Potentially solveable by letting CFS choose a RQ among the ones that are not in a Sleep State.

• Develop, implement, and test a scheme which detects potential overloads. We can currently only detect if a CPU has an idle time of naught, but are unable to detect if the load actually exceeds the capacity. One potential solution would be to add a metric akin to a soft deadline to every task, which would detect how long a task has to wait for CPU time. If the average detected waiting time for the whole RQ on exceeds a certain threshold, then the RQ would be flagged as overloaded.

• Confirm whether or not using a Governor in conjunction with OGS will muddle the `idle_time` readings. If this turns out to be the case, then figure out if polling the current CPU frequency is a solution.
CONCLUSION

The premise for this thesis was the use of a PID controlled power manager to improve the energy efficiency of a Linux-driven cloud system. We covered some basic Control Theory relating to PID controllers and explained how it relates to the proposed PID power manager. We also quickly covered how the power manager is meant to work in terms of how it will influence the system to achieve the control objectives. This is necessary as the work performed herein will be an autonomous extension of the aforementioned power manager in a cloud system.

We investigated the Linux kernel with the intention to identify any potentially pre-existing and implemented functionality related to energy efficiency. Some functionality was found to already be in place and its suitability for our planned implementation was evaluated. We found it to be insufficient for our purposes and hence a number of changes were proposed. The changes require the kernel code to be tweaked, therefore we isolated the code related to the changes we have proposed, and explained in detail how it works in the current kernel. Then we went on to explain how we want to implement our proposed changes as well as giving a detailed explanation of the changes already put in place both in terms of code and in terms of intended functionality. Not all of the proposed changes were implemented during this thesis project, and we have pointed out which ones are still missing. The proposed changes amount to a new scheduler we have decided to call the Overlord Guided Scheduler.

While the implementation is currently incomplete, it is possible to run simulations using what is already in place and compare the results against an
existing scheduler. All results were gathered from simulations using simulated systems. None of the results are from real-life cases, as the proposed scheduler would have to be complete before it can be safely used in such cases. The results were produced for a number for different architectures and parameters, and analysed in Excel on a per-architecture and parameter set basis. We showed that OGS can provide improvements to energy efficiency compared to CFS. OGS used fewer CPUs in all cases, and often with a negligible drop in performance. The simulations also showed that OGS is not quite complete, and illustrated a few issues that had not been considered, but are now listed under Further Work. The conclusions drawn from these results indicate that the proposed scheduler requires more work, but are otherwise promising with regards to energy efficiency. At the time of writing we are unable to produce concrete results regarding the actual energy efficiency of a cloud system employing a PID controlled power manager assisted by the proposed scheduler.
BIBLIOGRAPHY

[1] Efficient servers, a project conducted within the eu-programme intelligent energy europe. Online.


Introduktion

En snabbt växande befolkning har lett till ett allt större behov av elektricitet, medan en ökande miljömedvetenhet sätter allt högre krav på energieffektivitet i exempelvis datorsystem. För att inte tala om att en lägre strömförbrukning är mer ekonomisk.


Schemaläggning och strömförbrukningshantering i den nuvarande Linuxkärnan

Utvärderingen av Linuxkärnan påbörjas med en undersökning av de tillgängliga schemaläggarna och deras funktionalitet. I princip finns det i dagsläget bara två schemaläggare i kärnan, eftersom den gamla O(1)-schemaläggaren helt ersatts med Completely Fair Scheduler (CFS). Den andra schemaläggaren i kärnan är realtidsschemaläggaren, som, vilket namnnet antyder, sköter schemaläggning i realtid. Eftersom realtidsschemaläggning inte är av intresse i det här skedet kommer fokus att ligga på CFS.

CFS kan konstateras besitta en mängd önskvärda egenskaper. Den har till exempel gjort sig av med den diskreta körtidsindelningen som de tidigare schemaläggarna använt sig av, vilket förminskar tiden en processor (CPU) vilar i onödan för att en exekvering inte fyller hela körtidsindelningen. CFS har också gjort sig av med körköer och använder sig istället av ett röd-svart träd (RBT). Trädet är sorterat enligt i hur stort behov en process är av att få tillgång till processorn. Då ett program fått köra uppdaterar det sin körtid i enlighet med hur lång denna tid var och, i det fall det fortfarande är i behov av körtid, läggs sedan tillbaka i trädet. Eftersom trädet är självsorterande kommer den nyligen exekverade processen att flytta det till ställe i trädet som motsvarar dess nya körtidsbehov. På detta sätt håller schemaläggaren enkelt koll på vilken process som bör köras till näst.


Eftersom så gott som alla processer är ändliga kommer både antalet och fördelningen av dem att förändras medan ett system körs. I de fall ett system har

Linux är dock inte helt oanvändbart i sin nuvarande tappning. En användbar funktionalitet som redan finns är de så kallade Guvernörerna (Governors). Då de är i användning ser de till att CPU-frekvensen är proportionell mot den momentana arbetsbördan. Även om pluralformen för guvernör används här bör det antas att det är en samlingsterm, då endast en guvernör kan användas åt gången, men fungerar i mångd och mycket på samma vis som de andra guvernörerna. Att justera frekvensen i förhållande till arbetsbördan leder vanligtvis till att en CPU förbrukar en mindre mängd ström medan den fortfarande till synes sköter sina uppgifter lika fort och smidigt. För att åstadkomma detta håller guvernörerna koll på hur länge en CPU exekverar något och hur länge den vilar och sparar dessa värden i en struktur. Dessa värden kontrolleras med ett godtyckligt samplingsintervall och om vilotiden är längre än ett godtyckligt tröskelvärde konstaterar guvernören att frekvensen kan sänkas. Höjningen av frekvensen sköts i omvänd ordning [14][16][5][6][8][18][7][2][13][12][9].

**Kontrollerad strömförbrukningshantering**

PID-regulatorer förknippas vanligen med ett maskineri eller dylika koncept där det finns något konkret och uppenbart att kontrollera. Farthållaren i en bil t.ex. använder sig av en regulator, vars uppgift det är att ta den önskade hastigheten som referensvärde och använda detta i samband med den nuvarande hastigheten för att beräkna hur mycket gasreglaget bör justeras för att uppnå målsättningen.

Om man förenklar något kan man säga att en PID-regulator löser en andra gradens differentialekvation med avseende på ett felvärde. Felvärdet är differensen mellan referensvärdet och det nuvarande uppmätta värdet. I fart- hållarexemplet är referensvärdet den önskade hastigheten och det uppmätta värdet är den resulterande hastigheten. Utgående från detta fås:


\[ y = K_i \int_0^t e(\tau)d\tau + K_p e(t) + K_d \frac{d}{dt} e(t) \]  

(8.1)

där \( y \) är utvärdet till farthållaren, \( e() \) är felvärdet - \( e = r - y \) där \( r \) är referensvärdet - och \( K \) är förstärkningskonstanter specifika för de enskilda operationerna i ekvation 8.1.


Schemaläggningsrelaterade förändringsförslag


För att välja RQ i enlighet med kraven bör OGS ha koll på belastningen i varje enskild CPU. Lyckligvis behöver detta inte skapas skilt, utan det är möjligt att använda sig av samma information som guvernörerna samlar in. Denna information inkluderar alla bidrag till exekveringstid på en CPU, vilket gör det möjligt att utgående från detta fatta ett beslut om huruvida en CPU är fullt belastad eller inte. En fullt belastad CPU kommer inte att ha någon vilotid, medan en CPU med tillräckligt stor vilotid kan hantera fler processer. OGS kommer alltså att schemalägga nya processer på samma CPU tills dess vilotid är så gott som noll.

Belastningsbalanseringen är i skrivande stund inte klar och implementerad ännu, men det finns dock en idé om hur detta kunde lösas. LB-procedurens
beslut kommer att basera sig på samma information som RQ-valen. Proce-
duren håller koll på vilken RQ som är mest belastad, men med kvarvarande
kapacitet, samt vilken RQ som är minst belastad. I mån av möjlighet flyttas så
många processer som den mest belastade RQ:n kan hantera från den minst
belastade. Då denna procedur utförs periodiskt kommer antalet belastade
CPU:n att hållas möjligast litet.

Sätten som RQ-valsalgoritmen och LB-proceduren fungerar på är relativt
viktiga för att PID-regulatorn ska kunna utföra sitt jobb obehindrat. Då dess
primära uppgift delvis består av att stänga av vilande CPU:n är det viktigt
att enstaka processer inte exekveras på en CPU här och en CPU där. OGS har
därför som uppgift att assistera regulatorn genom att samla alla processer i
systemet på möjligast få CPU:n utan att offra alltför mycket i form av uppfattad
prestanda.

Verktyg, experimentation och testning

Eftersom schemaläggning är ett av de mest komplexa koncepten i linuxkärnan
och ett av de mer kritiska ur en prestandasynvinkel, är det otroligt utmanande
att skriva en felfri schemaläggare. Tar man ännu installerings- och byggtiden
för en ny kärna i beaktande kommer man ganska fort fram till att detta inte är
något man gärna gör direkt i systemet man för tillfället använder. Till all lycka
finns det nuförtiden verktyg som LinSched.

LinSched tar sig runt de ovan nämnda problemen genom att migrera sche-
maläggningssubsystemet till "user space". På så sätt är det i princip möjligt att
köra tester på den planerade schemaläggaren i något av en nerbantad virtuellt
kärna. LinScheds version av kärnan saknar dock en hel del funktionalitet och
verifierar t.ex. inte läsning, så schemaläggaren måste ändå i något skede testas
i en full kärna, men på det här sättet kraschar man inte hela maskinen om man
 gjort ett fel.

En för OGS viktig funktionalitet visade sig saknas i LinSched-kärnan, näml-
gen informationen som guvernörerna bidrar med. För att komma runt denna
begränsning lades falska uppdateringar, som bidrog med exekveringstider
för processorer, till i LinScheds källkod. Detta innebar att informationen
på vilken OGS baserade sina RQ-val inte var helt korrekt, men det var
åtminstone möjligt att verifiera dess beteende och efter lite finjusteringar även
att jämföra dess prestanda med CFS. En annan funktionalitet som saknades
var systemtiden. Både OGS och guvernörerna använder sig av denna för att
hålla koll på hur lång tid som förflutit sedan det senaste beslutet gjordes. Detta
löstes genom att plocka in systemtiden från den egentliga kärnan.
Simuleringarna kördes sedan för fem olika topologier med ett stort antal olika simuleringsparametrar för både CFS och OGS. Resultaten jämfördes sedan i Excel för de olika topologierna och parametrarna. Utgående från dessa gjordes sedan ett antal beräkningar, vilkas resultat ritades som grafer för att enklare visualisera dem. I dessa grafer jämfördes exekverings- och fördröjningstider, processorernas allokeringstid, antal processer per CPU, skillnaden i exekveringstid per process samt exekveringstidsvoterna per process.

Då simuleringarna inte skedde i verkliga system kunde strömförbrukningen inte mätas och jämföras för schemaläggarna. För att kunna göra en approximerad jämförelse av de två gjordes antagandet att varje belastad CPU innebär en konstant förbrukning på 10 W.

### Resultat

I resultaten kunde påvisas att OGS i alla fall använder färre CPU:n än CFS. OGS påvisar dock även ett antal buggar samt att LB fortfarande saknas. Då detta tas i beaktande kan man konstatera att om dessa resultat gäller även i en full implementation i en komplett kärna, så är OGS mer energieffektiv än CFS med en mycket liten negativ inverkan på prestanda.

### Vidare arbete

Som det har nämnts ett antal gånger är OGS inte klar än. En del funktionalitet saknas och en del borde ännu kontrolleras efter buggar och alltihop borde testas i en simulator som t.ex. Simics som kan utnyttja en full kärna. Det hittills identifierade behovet av vidare arbete är:

- Testa schemaläggaren i en full kärna.
- Lägg upp ett test där den faktiska strömförbrukningen kan mätas.
- Implementera resten av OGS och testa detta.
- Utveckla, implementera och testa ett sätt att upptäcka möjliga överbelastningar. Dessa kan ske om en process visar sig kräva en längre exekveringstid än uppskattad.
- Utred och testa huruvida användningen av en guvernör i samband med OGS fungerar med de nuvarande algoritmerna. Det finns en möjlighet
att guvernörens justeringar av CPU-frekvensen leder till att OGS får felaktiga värden.

**Slutsats**

Detta arbete har baserats på idén att en PID-reglerad strömförbrukningshantering kan användas för att förbättra energieffektiviteten i ett linuxdrivet molnsystem. Grundläggande reglertekniksteori relaterad till PID-regulatorer förklarades samt sambandet mellan denna teori och den planerade strömförbrukningshanteringsregulatorn. Hur regulatorn ska nå sin målsättning har även förklarats, vilket är nödvändigt eftersom OGS kommer att fungera som en autonom förlängning av strömförbrukningshanteringsregulatorn i ett molnsystem.


Även om implementationen för tillfället inte är fullbordad var det möjligt att köra simulationer med den funktionalitet som finns på plats och att jämföra de producerade resultaten med den existerande schemaläggaren. Inga resultat har i det här skedet producerats med en full kärna på ett verkligt system, då schemaläggaren bör vara fullständig föröver den kan användas i ett dylikt system. Slutsatserna dragna från de producerade resultaten är att OGS verkar kunna förbättra energieffektiviteten i Linux, men att en mängd arbete kvarstår föröver schemaläggaren kan påstås vara fullständig och klar för en verklig tillämpning. Problem och fortsatt arbete har listats i kapitlet om framtida arbete.
In file:
    kernel_source_code/arch/x86/Kconfig

- Added config option and other relevant bits for Overlord.
  - Option is dependent on SMP, as it is pointless in systems
    with only one cpu
  - Default choice is N
  - Also added a quick help description

- The same thing should be added in all arch-related Kconfigs

In file:
    sched.c

- Put other OGS-related definitions, i.e. those not of the form
  above, behind an #ifdef for the same reasons

- Added files to the include list
  - stdio.h for print functions
  - linux/fake_kstat.h in case we need the fake time
    accounting here (behind #ifdef)

- Option to print location from various functions in order to
  keep track of where we are.
  - #ifdef PRINT_LOCATION in:
- schedule()
- scheduler_tick()
- resched_task()
- set_task_rq()
- __set_task_cpu()

- Option to print capacity
  - #ifdef PRINT_CAPACITY in:
    - schedule()
      - Prints the returned CPU’s ID and its idle time

- Option to turn off load balancing
  - #ifndef NO_LOAD_BALANCE
    - Removes trigger_load_balance(), i.e. the first function in the LB chain, at compilation time

- Added functions: update_fake_*_time()
  - Updates the faked busy time accounting
  - Updates parts of the struct in fake_kstat.h depending on which of the functions we’re calling
  - Most of the updates use a static and arbitrary number
  - update_fake_busy_time uses the sum_exec_runtime value to give us something that varies
    - ((struct task_struct*)current)->se.sum_exec_runtime
  - This is the same runtime value linsched uses in its reporting function
    - Called from:
      - wake_up_new_task()
      - try_to_wake_up()

In file:
  sched_fair.c

- Added #ifdefs to various places to make swapping between CFS and OGS easier
  - CONFIG_SCHED_OLORD
  - CONFIG_OLORD_GROUP_SCHED
  - In most cases we use the CFS code in this file

- Added #ifdefs for when linsched is no longer used

- Added files to the include list
- stdio.h for print functions
- linux/fake_kstat.h in case we need the fake time accounting here (behind #ifdef)

- Added definition of a struct array
  - struct colonel_stat cstat[LOTS]
  - Used for busy time accounting
  - LOTS = 100, defined in fake_kstat.h

- Added functions: update_fake_*_time()
  - Updates the faked busy time accounting
  - Updates parts of the struct in fake_kstat.h depending on which of the functions we’re calling
  - Most of the updates use a static and arbitrary number
  - update_fake_busy_time uses the sum_exec_runtime value to give us something that varies
  - ((struct task_struct*)current)->se.sum_exec_runtime
  - This is the same runtime value linsched uses in its reporting function
    - THESE FUNCTIONS WERE MOVED TO SCHED.C
    - Now in:
    - wake_up_new_task()
    - try_to_wake_up()

- Added function: what_is_idle_time()
  - Calculates the amount of idle time on a CPU.
  - Uses the struct available in include/linux/kernel_stat.h
  - This struct is also used by the governors
  - All update functions for this struct should be included in the original kernel code
    - If we’re using linsched then the struct is "faked" as the update functions do not works
  - Use the update_fake functions mentioned above to keep track of the idle time
  - Does not correspond entirely to reality
    - Changed the CPU ID check
  - We need one as the cpu value is an unsigned int in some places
  - If the ID is out of range, return idle time = 0 which will exclude the CPU from the search
    - This is a temporary solution

- Added function: find_busy_group()
  - Finds the busiest scheduling group with enough idle time
to be able to accommodate new tasks
- For more details see the function
- Changed how the SAFETY_MARGIN is noted
- We now have a separate variable here, which is incremented by one SAFETY_MARGIN per group member

- Added function: find_busy_sched_cpu()
  - Finds the busiest cpu with enough idle time to be able to accommodate new tasks
  - Calculates the average busy time contribution per task on the runqueue to figure out if adding another task is likely to cause an overload
  - For more details see the function

- Option to print location from various functions in order to keep track of where we are
  - #ifdef PRINT_LOCATION_FAIR in:
    - select_task_rq_fair()
    - find_busy_sched_cpu()
    - run_rebalance_domains()
    - rebalance_domains()
    - move_tasks()
    - balance_tasks()

- Option to print capacity
  - #ifdef PRINT_CAPACITY

- Option to print cpu choice
  - #ifdef PRINT_CHOICE

- Added fake time accounting to select_task_rq_fair
  - This is to make sure we actually update the kernel stats when a test is run
  - It may turn out we account for the time twice when running a simulation, although this is not really a problem since it should mean we need to use another cpu quicker which in turn makes it easier to see if everything works as intended

- Edited structure sd_lb_stats
  - Uses original structure when Overlord is not used
  - Uses elements with the same name for Overlord, but they are now of type cputime64_t
  - If we’re using Linsched, they’re still unsigned longs, but put
inside their own #ifdefs in case we have to change something
- Added elements when using Overlord
- total_idle, for tallying the total idle time for all groups in the SD
- this_idle, for tallying the idle time for this group
- unsigned longs when using Linsched, cputime64_t else
- Edited structure sg_lb_stats
  - Uses original structure when Overlord is not used
  - Uses elements with the same name for Overlord, but they are now of type cputime64_t
- If we’re using Linsched, they’re still unsigned longs, but put inside their own #ifdefs in case we have to change something
- Added CONFIG_SCHED_OLORD as an alternative condition before init_sd_power_savings_stats
  - We always want to use this with Overlord
- Changed conditional in: update_sd_power_savings_stats
  - If the local group is idle or completely loaded, then there is no need to do power savings balance at this domain.
- For Overlord, the capacity is measured in idle time and NOT how many tasks we’ve running. Thus the comparison is now performed against this_load instead of this_nr_running
- Edited function in: update_sd_power_savings_stats()
  - #ifdef conditional for Overlord
  - Inside #ifdef:
    - Changed conditionals so they now look at and update according to idle time. Was based on nr_running previously.
    - The function contains what is essentially two sets of similar conditions and value edits. This is because it makes the code easier to read and the #ifdef gets rid of the part we’re not using during compilation.
- Edited function in: update_sg_lb_stats()
  - Again, added a copy of this function’s internals for Overlord, which was subsequently edited
  - Usage governed by #ifdefs
  - Removed the local group bias
- Changed the load calculation to what_is_idle_time() from target_load()
- Removed the update related to weighted_cpuload(). We don’t use it anyway
- sgs->group_capacity is now calculated in the for-loop
- Removed a calculation adjusting the load by the relative CPU power of the group.
- Removed a call to update_group_power(). Does nothing relevant at this point.
- Group imbalance is flagged if:
  - min_cpu_load can fit in max_cpu_capacity
  - max_cpu_capacity is larger than avg_load_per_task
- Add more cases?
  - Added printf() after the for-loop to check if we can use "i" from the loop-variables
- If a usable value for NR_CPUS exists, then "i" could be set = NR_CPUS+1

- Edited function in: update_sd_lb_stats()
  - Again, added a copy of this function’s internals for Overlord, which was subsequently edited
  - Usage governed by #ifdefs
  - Removed : sds->total_pwr+= group->cpu_power
    we’re not likely to use it
  - Changed some values *this will get
- sds->this_load = sgs.avg_idle
  from: sds->this_load = sgs.avg_load
- sds->this_load_per_task = sgs.avg_load_per_task
  from: sds->this_load_per_task = sgs.sum_weighted_load
- Smaller this_load values now mean a higher load. May yet change the name of the struct element to this_idle
  - Changed some values *busiest will get
- sds->max_load = sgs.avg_idle
  from: sds->max_load = sgs.avg_load
- sds->busiest_load_per_task = sgs.avg_load_per_task
  from: sds->busiest_load_per_task = sgs.sum_weighted_load
- Smaller max_load values now mean a higher load. May yet change the name of the struct element
  - Changed conditions for setting the *busiest values
- Now: sgs.avg_idle < sds->max_load &&
  (sgs.group_capacity < SAFETY_MARGIN || sgs.group_imb))
- Before: sgs.avg_load > sds->max_load &&
  (sgs.sum_nr_running > sgs.group_capacity || sgs.group_imb))
- Purpose: Have the busiest group be one that is overloaded, where *group_leader is a busy NON-OVERLOADED group. This should mean we can up performance a bit, while still maintaining power efficiency

- Edited function in: fix_small_imbalance()
  - Again, added a copy of this function’s internals for Overlord, which was subsequently edited
  - Usage governed by #ifdefs
  - Removed variable inits from the original function. Turned out they weren’t needed
  - Removed the init calculations from the original function. Turned out they weren’t needed
  - *imbalance is now set if:
    - max_load is less than SAFETY_MARGIN and this_load is large enough to fit at least one task from busiest
    - max_load and this_load are reported in terms of idle time
    - this_load could fit at least one task from min_load
    - By leeching tasks off the least loaded every chance we get should lead to our being able to put more cpus to sleep

- Edited function in calculate_imbalance()
  - Again, added a copy of this function’s internals for Overlord
  - Usage governed by #ifdefs
  - Removed averaging of busiest_load_per_task. It already IS an average.
  - if (sds->group_imb) does nothing at the moment
  - if (!sds->group_imb) changed:
    - Sees if busiest is overloaded and if so; calculates the amount
    - Sets load_above_capacity = 0 if busiest is not overloaded
      - Changed the way *imbalance is determined we should pull from the target.
    - We see if busiest is not overloaded and calculate the number of tasks we could pull to it
    - We calculate the number of tasks we could pull to leader
      - If we can pull 1 or more tasks to busiest, do it
      - If we cannot pull to busiest, but can pull 1 or more tasks to leader, do it
      - Otherwise, pull to this
    - If busiest turned out to be overloaded, just check leader and this to see how much to pull
    - Make sure imbalance is at least 1
- If not, then try fix_small_imbalance()
- On returning from this function,
  imbalance = NUMBER OF TASKS TO PULL
- Commented scenario that seemed to be specific for CFS
  if (sds->max_load < sds->avg_load)
- Edited function in find_busiest_group()
  - Removed conditions for when imbalance does not exist from
    the POV of this_cpu
  - Imbalance is within the specified limit
  - There is no busy sibling to pull from
  - This group is more busy than the average business at this
    sched_domain
  - New (and re-used) conditions for when imbalance does not
    exist from the POV of this_cpu
  - this_cpu is not the appropriate cpu to perform load balancing
    at this level
  - This group is the idlest group
  - This group is the busiest group and can’t take on any new tasks
  - Busiest is not Overloaded

- Edited function in check_power_save_busiest_group()
  - Again, added a copy of this function’s internals for
    Overlord, which was subsequently edited
  - Usage governed by #ifdefs
  - Introduced variable int power_save_pull
    - The number of tasks we can pull from busiest or idlest
    - Integer so we don’t have to worry about rounding
    - Never greater than the number of tasks running on the
      target queue
  - Changed conditions for returning 0. We return 0 if:
    - If this != group_leader
    - If group_leader = group_min
    - If leader can’t take on any more tasks
  - Changed what is due to happen if we return 1.
    - If busiest_group_capacity > SAFETY_MARGIN (i.e. is not
      overloaded)
    - Calculate how many tasks we can pull from group_min to leader
    - Make sure we don’t try to pull more than min_nr_running
    - Set sds->busiest = sds->group_min (because the balancing
      logic always pulls from busiest)
    - If busiest_group_capacity < SAFETY_MARGIN (i.e. IS
overloaded)
- Calculate how many tasks we can pull from busiest to leader
- Make sure we don’t try to pull more than busiest_nr_running
  (this shouldn’t be possible)
- Just return 1, since sds-> busiest already points at the
  queue we want to pull from

- Edited function in find_busiest_queue
  - Again, added a copy of this function’s internals for
    Overlord, which was subsequently edited
  - Usage governed by #ifdefs
  - Added variable time_now
    - For Linsched type = unsigned long
    - Else type = cputime64_t
  - Removed variable power
  - Changed values for:
    - Capacity now it is = what_is_idle_time(cpu, time_now)
    - Workload (wl) now it is = time_now - capacity
  - Changed conditions for skipping to the next increment
    - If nr_running < 2 :: continue
  - This is because an RT-task might use loads of CPU time, but
    moving it somewhere else, would not improve on anything
    Not to mention that its extremely high avg_load_per_task
    value would most likely mean no RQ can take it

- Edited function in load_balance()
  - Added printf to see when we end up in this function
    - Usage governed by #ifdef

- Edited function in load_balance_fair()
  - Again, added a copy of this function’s internals for
    Overlord, which was subsequently edited
  - Usage governed by #ifdefs
  - Added printf to see when we end up in this function
    - Usage governed by #ifdef
  - Removed calculation update_h_load(busiest_cpu)
    - Do not need/use it
  - Removed calculation rem_load * busiest_weight &
    div_u64(rem_load, busiest_h_load + 1)
    - Do not need/use it
  - Removed calculation moved_load * busiest_weight &
    div_u64(moved_load, busiest_h_load + 1)
    - Do not need/use it
- Changed variable type
  - u64 to unsigned long for rem_load
  - u64 to unsigned long for moved_load

- Edited function in balance_tasks()
  - Added a copy of most of the function’s parts.
    - Only had to change the bits relating to the moved load at the start of the loop

  - Usage governed by #ifdef
  - Change conditions for invoking continue
  - Not using load weights, so no point in having
    (p->se.load_weight >> 1) > rem_load_move
    - Now only invoked if !can_migrate_task()
  - Change decrement of rem_load_move
    - Now decremented by 1, since *imbalance is in number of tasks to move

In file:
fake_kstat.h

- New file
  - Placed in /include/linux
- Emulates the necessary parts of kernel_stat.h
- Contains the busy time accounting function prototypes

In file:
sched.h

- Added #ifdefs to make swapping between CFS and OGS easier
  - CONFIG_SCHED_OLORD
  - CONFIG_OLORD_GROUP_SCHED
- No big changes here. We use CFS-code in all cases
LINSCHED CHANGE LOG

In file:
  Makefile
  - Added definition flags for OGS
    - -D CONFIG_SCHED_OLORD
    - -D CONFIG_OLORD_GROUP_SCHED

In file:
  basic_tests.c
  - Added #ifdefs to various places to make swapping between CFS and OGS easier
  - Added ridiculously lax result validation requirements to all tests
  - Added the result printing function to all tests. They are now printed regardless of the outcome
    - linsched_print_task_stats()
  - In test_trivial_bal()
    - Task creation is now in a for-loop
    - We create one task at a time and run it for 5ms
    - We create one task per cpu
    - The test is then run for an additional (TEST_TICKS-(count*5))ms
    - Creating tasks like this will give the kernel stats a chance to update before the next task is created
  - In test_basic_bal1()
    - A similar change to task creation
- Create 5 sleep/run tasks and run sim for 500ms
- Do this for each CPU
- Run the test to its conclusion, i.e.
  \[(\text{TEST TICKS}-(\text{count}\times 500))\]ms
- This gives us quite a few tasks to schedule, but fewer than we’d need to fill up all cpus (I think), which should help in identifying potential problems
- In test_basic_bal2()
  - A similar change to task creation
  - Create 5 sleep/run tasks and run sim for 500ms
  - Do this for each CPU
  - Run the test to its conclusion, i.e.
    \[(\text{TEST TICKS}-(\text{count}\times 500))\]ms
  - This gives us quite a few tasks to schedule, but fewer than we’d need to fill up all cpus (I think), which should help in identifying potential problems
- In test_bal1()
  - A similar change to task creation
  - Test includes cpu-bound tasks, so create these first and run for 100ms
- Overlord does not take cpu affinity into consideration at this point
  - Create 5 sleep/run tasks and run sim for 500ms
  - Do this for each CPU
  - Run the test to its conclusion, i.e.
    \[(\text{TEST TICKS}-(\text{count}\times 500)-100)\]ms
  - This gives us quite a few tasks to schedule, but fewer than we’d need to fill up all cpus (I think), which should help in identifying potential problems

In file:
  - Added fake_kstat.h to the includes
    - Include is dependent on the definition of CONFIG_SCHED_OLORD
  - Added fake busy time accounting function calls to linsched_announce_callback()
    - Usage dependent on the definition of CONFIG_SCHED_OLORD
  - Added a printf to linsched_print_task_stats()
    - Now prints an empty line before the full stats print. Makes reading the results easier