Bachelor Thesis: A Fair $\mathcal{O}(1)$ High Throughput CPU-Scheduler for Linux (HTFS)

Markus Pargmann
6535177

November 1, 2011

presented to

Prof. Dr.-Ing. André Brinkmann

Abstract

The server market is continously growing to fullfill the demands of cloud computing, internet related servers, like HTTP or Email server, high throughput computing and much more. To reach the highest possible ressource utilization, modern operating system kernels are highly optimized. This also is the case for the Linux CPU scheduler. But especially for servers the Completely Fair Scheduler has some performance flaws.

In this bachelor thesis a new CPU-scheduler design is proposed. The High Throughput Fair Scheduler (HTFS) is a multi-queue design, which is able to fullfill $\mathcal{O}(1)$ limitations. To assure fairness to all tasks this classical queue-design is extended with virtual runtimes. Through a non-strict fairness HTFS can work with less task switches, which results in higher throughput. HTFS, aimed at high scheduling speed, fairness and throughput, is able to compete with the Linux version 2.6.38 CPU-scheduler.

Contents

1 Introduction

In the last years, the trend of virtualization has been growing. It is a good way of increasing power efficiency and offering the possibility to easily migrate running systems to other hardware. The successor of virtualization is cloud computing, which allows transparent scaling of computing power. This is done by creating new processes or new virtual machines on demand. A normal service or application does not use constant ressources at every time. For example a web server needs much ressources in the evening but not at the morning. Through different demands of services, applications and companies, virtualization and cloud computing reach much higher ressource usage and power efficiency than servers with only a few different services.

Additional to this trend, the computing power of processors and servers is increasing. The big chip manufacturers only produce multi core server processors anymore. In modern processors, instructions per clock are still increasing. At the same time – especially in the high performance computing market – the share of Linux continuously grows. In 2011 the share of Linux in the top 500 supercomputers reached 91.2% [1]. The Linux webserver market share was 41.02% [2] in 2009.

The currently used Completely Fair Scheduler (CFS) in Linux version 2.6.38 is strictly fair. It is a tree based design with virtual runtimes, that are used as a key to sort tasks within the tree structure. The time complexity is $\mathcal{O}(\log n)$. Independent of time complexity, CFS is relatively time consuming, which leads on heavy loaded servers to a low utilization of the CPU. Especially in the area of high performance computing, better usage of the CPU can produce a reasonably higher throughput. Currently there are no other CPU schedulers aimed at the server market.

A 1% increase of performance for supercomputers is a lot. For a server centre with 100 servers, this could offer the computing power of 1 additional server without any expense. In time, this would be an increase of 14.4 minutes of available computing time per day, or 3.6 days per year, which the system could use. This would further increase the efficiency of cloud systems.

To increase the speed of a CPU scheduler, there are three important factors. First, the scheduler should be independent of the number of running processes, so it has to be within the $\mathcal{O}(1)$ boundaries. The $\mathcal{O}$ notation does not reflect any constant time consumption, which leads to the second factor, a fast scheduling code. Beside those code based improvements, the system speed can be increased by better scheduling decisions and less task switches.

In this thesis the High Throughput Fair Scheduler (HTFS) was developed featuring higher CPU utilization through a faster scheduling. It uses a classical multi queue design. The design is able to increase the speed of the scheduling code by decreasing the necessary operations for normal scheduling. It uses adaptive timeslices that can increase to a maximum of one second, which can massively decrease the number of task switches. Because of the inability of the design to properly treat interactive tasks, HTFS treats them in a special way, unlike CFS where interactive tasks automatically benefit from the tree design.

HTFS is extended by virtual runtimes to make the scheduler fair. This is a more rough fairness in contrast to the strict fairness of CFS. So HTFS increases the throughput by loosening the strict fairness of CFS. The loss of strict fairness should not be noticeable by any user or program, but makes it possible to avoid sorting the tasks, which can be time consuming.

CFS's implementation has very accurate calculations. HTFS tries to make the scheduling as fast as possible by replacing calculations with good estimations that are cheaper to compute. Also continous values are mapped to discrete levels to reduce the number of divisions and comparisons.

For workloads, which trigger very few task switches, HTFS won't increase the performance. But workloads with a very high number of task switches should perform 2% better than CFS. This performance benefit could scale with the number of CPUs because HTFS only modifies the per-CPU structures and algorithms. This leads on a dual core system to a possible better performance of 3.8%. So HTFS is a scheduler design that could be interesting for servers in the future.

2 Basics

Every operating system has some sort of kernel, a central component that manages everything between the application level and the hardware. Through the ability of multitasking of all operating systems today, the ressources managed by the kernel become shared ressources. Those have to be managed by the kernel, so there are many different areas where the kernel needs schedulers, e.g. IO-Scheduler. The processor is also a ressource with an own scheduler.

The Linux kernel scheduling architecture consists of 3 main parts, the scheduling code itself and two data structures for the representation of the system's hardware hierarchy and to store tasks in runqueues dependent of their taskgroups and CPU they run on.

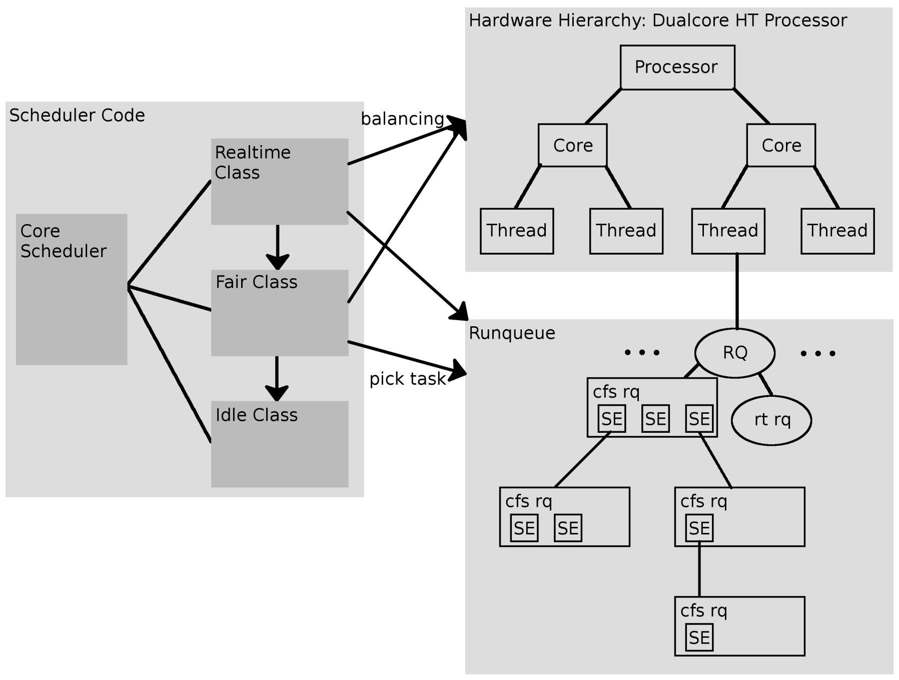


Figure 1: Diagram of the scheduling architecture in the Linux kernel. SE = Schedulable Entity, RQ = Runqueue, RT = Realtime, CFS = Completely Fair Scheduler, HT = Hyper Threading

In the following subsections, I will describe all visible and invisible parts of the architecture presented in Figure 1.

2.1 Interactive Task

Every system has processes with different workloads. On the task level we can distinguish between two main types. Those are CPU bound tasks and interactive tasks. The first category describes all the tasks using only the CPU. The second category contains all tasks that are using input/output operations or other operations like locking, which cause the task to switch the state between running and blocked. Those tasks often cause new scheduling requests, before the end of the task's timeslice. There are two main reasons for early scheduling requests. The first is the state switch from running to blocked, which forces a schedule. The task does not wait for the event on the runqueue, instead mechanisms trigger the task to be enqueued to the runqueue again, after the desired event occures. This is probably used if it takes a long time for the event to occur. The second reason is busy-waiting. Busy-waiting means the task remains in the "running" state while waiting for the event. In practice the task is checking for a condition and triggers a reschedule if the condition was not met yet. This concept is mainly used if the event occurs fast.

For interactive tasks, waiting for an event dominates the runtime of the whole process. To reduce the runtime of those interactive tasks, it is important to reduce latency of an interactive task getting CPU time after awaking. This also improves the responsiveness of the system, which is really important, especially for system with a graphical interface.

2.2 Scheduling Class

The Linux kernel scheduling code has a core and several scheduling classes. The core scheduler wraps all important functions around the scheduling classes. For example, the scheduler timer interrupts are managed by the core code, while scheduling decisions are mainly handled in the scheduling classes. The rest of the kernel does not know about the classes, so they call the scheduling routine via the core scheduler.

A scheduling class implements a scheduling class interface with all the necessary functions to create an own scheduler. Although the abstraction is really good, it is not completely consistent. Some balancing routines are not attached to the scheduling classes. For better performance, some scheduling class specific code is integrated in the core scheduler. Class specific data is embedded in the general core scheduler structures. So at the end it is a design, which offers high performance while seperating the code in different classes.

In the Linux kernel version 2.6.38, there are two main scheduling classes, the realtime class and the fair class. Both classes try to accomplish different scheduling characteristics. The realtime class behavior is more predictable for realtime applications. They need to know when they can expect a special task done or that their task gets a guaranteed timeslice. This class lacks any special treatment of interactive tasks. That is what the fair scheduling class is for. It serves fair treatment of all tasks and a good behavior for interactive tasks. The class a task belongs to is stored through the priority of a task. Priorities from

0 to 99 are realtime priorities, 100 to 139 are priorities of the fair scheduling class.

In most cases, a schedule request is initiated by a scheduling class or by a task that runs in kernel space and wants to leave the CPU for the moment. Often a schedule request ends in a new task running on the CPU but this is not true for all requests. Sometimes the currently running task is the only task that can run at the moment or the scheduling class decides to leave the task on the CPU. On receiving a schedule call, the core scheduler checks all classes for runnable tasks. First it calls the "pick next task" function of the realtime class. If this returns no task, the fair class is asked for a task. If both classes have no runnable tasks, the third class, "idle class", is called, which always returns an idle task. This is no real class as it does not have any real tasks. It is just a nice possibility to embed the automatic scheduling of idle tasks.

There are some other functions for scheduling classes to implement. For example there is a tick function in which the class should check if the currently running task had enough time on the CPU. It is required for the maintenance of the scheduling class specific data structures to have operations like enqueue, dequeue, put previously running task and pick next task to run. More advanced interface functions are runqueue online and offline notifications, task fork notification, task wakeup, check preemption and some more. By providing this wide interface, the scheduling class is able to react on nearly everything. [3] [4]

2.3 Runqueue

A runqueue is a data structure that stores tasks with the state "running". Every CPU in the system has an own runqueue. All runqueues together hold all tasks, which are in the state "running". Some years ago Linux had only a single runqueue for all CPUs in a system. The runqueue had only a single lock to prevent multiple threads modifying data and leave the runqueue inconsistent. This produced lock contention, which would cost much time in modern many core systems. Today, the multi runqueue design prevents this lock contention. Another benefit of this design is the ability to cache important datastructures in the CPU local caches. Only for some balancing functions, the local caches have to be synchronized. Both aspects increase the speed of the scheduler code. Through multiple runqueues there is a new part of the scheduler, which deals with load balancing, which is described later in the "Load Balancing" section.

Every scheduling class has an own datastructure embedded in the general runqueue structure to store those class specific tasks, see Figure 1. There is the top runqueue structure, which holds a "rt rq", the realtime runqueue, and a "cfs rq", the completely fair scheduling runqueue. The "cfs rq" is the runqueue of the current fair scheduling class implementation, which is CFS. If the scheduler calls the "pick next task" functions of the scheduling classes, those are searching for a suitable task in their specific runqueue structures. In Figure 1 I only presented the "cfs rq" in detail because the "rt rq" is of no interest for this bachelor thesis.

The "cfs rq" has multiple schedulable entities (SE). It is very important to differentiate between a task and an entity that is scheduled. There are two types of schedulable entities. The first one symbolizes a normal task. The second type is an entity, which has another “cfs rq” structure inside and represents a taskgroup. This is important for the taskgroup feature, which is described later.

To get a task for the “pick next task” call, the scheduling class simply chooses a SE from the cfs rq. As long as the picked SE is a taskgroup, a new SE is picked from the included “cfs rq”. The current fair class implementation, CFS, uses a red black tree for storing the SEs in the “cfs rq”. [3]

2.4 Hardware Hierarchy

First I will define some naming conventions for this work regarding the hardware. A processor is a whole chip with pins that is plugged into a socket on the mainboard, so the complete unit. In the following, I will use the term “CPU” as an unit that can run a single thread. A dual core processor with hyperthreading for example, provides 4 distinct CPUs in this terminology. So a hyperthreaded core has two CPUs, even if a hyperthreaded core does not have all processing units twice.

Every computer has a natural hierarchy. On single thread processors, this hierarchy collapses to a single level. With hyperthreading, chip multiprocessing or symmetric multiprocessing, the hierarchy has multiple levels. This hierarchy is represented in the Linux kernel scheduler. One level of the hierarchy is represented by a “scheduling domain structure”. This structure stores some information flags for hardware aware load balancing. As an example, the task migration on the same core between two CPUs is much cheaper than migration between two processors. There are several flags and a migration cost value for deciding better on task migrations. This information is initialized by the hardware architecture specific code.

To speed up the scheduling domain access, there is not a single hierarchy tree in the Linux kernel. Instead on every CPU the hierarchy path to this CPU is stored. No other CPU is using this hierarchy path. The siblings in the hierarchy tree are represented by scheduling groups, which span a subset of all available CPUs. The scheduling group of the next higher scheduling domain is a superset of the scheduling group in the scheduling domain below. So the union of scheduling group’s sets of CPUs on one scheduling domain level, is always a full set of all CPUs. The intersection of two different scheduling groups of the same scheduling domain level is empty. [5] [6]

2.4.1 Load Balancing

Through the multi runqueue design there is an important new operation: the load balancing. It tries to achieve an equal level of load on all CPUs. This is not completely accurate, like a single runqueue with all the tasks in a system would be, because load balancing only is able to react on load changes. The single runqueue design always can be fair. There are two different load balancing mechanisms, the normal and the active one load balancing. The first one is called if the load is not completely in balance. It tries to migrate some tasks between the CPUs to reach a good balance. If this does not work and the imbalance remains too high, the active load balancing is called to migrate tasks agressively.

Both load balancing routines start to find suitable migrations on the lowest level in the scheduling hierarchy, going up the hierarchy. All task migrations are done within a scheduling group, which prefers task migrations on the lower levels of the hierarchy.

The decision for a task migration is made with help of many metrics. Every task has a weight value, which denotes the task's load. The meaning of weight is explained in the "Priority" section (2.5). First, the weight of a task is not allowed to create a new imbalance, as migration would not make sense in that case. For minimum energy consumption, if wished, the balancing tries to hold all tasks on the CPU, until the CPU's capacity is reached. This allows other CPUs to run on lower frequencies or even being powered off. The balancing also tries to keep new tasks together, which have pipes to each other. Those processes should have higher throughputs when located on a single CPU because of the local caches that are transparently used for the pipe communication. Another metric is the migration cost for a task on a specific domain level. This is dependent on the architecture and on the domain level where the imbalance occured. The last metric is cache hotness. For example if a task just ran on a CPU, the cache is full of data from this task. A migration would be very expensive because the cache of the new CPU has most likely no data of this task. I won't describe the load balancing methods any further here because it is not important for HTFS. [3] [6]

2.5 Priority

Linux has 140 priorities for tasks to differentiate the importance of tasks. The priorities are related to different amount of runtime. The priority range is split into two parts. The first 100 priorities are realtime priorities, which are managed by the realtime scheduling class. Priorities 100 to 139 are managed by the fair scheduling class. Those priorities are also known as the NICE levels from -20 to 19.

The realtime scheduler is unimportant for this bachelor thesis, so I am concentrating on the fair scheduler here. The NICE levels are translated to weight values. Table 1 shows the NICE levels with their according weight values. A low NICE level means the process gets much runtime.

The runtime of two processes with a priority difference of one, have about 25% different runtimes. A process that switches in a situation with other processes from a priority to the next higher one gains 10% because the runtime shares are newly divided. For example process A with priority 0 and process B switching from priority 0 to 1. Before switching, A and B have the same share of the available runtime, for example 50 each. Increasing the priority of B the share of the runtime is recalculated. Now process B has to get 25% more runtime. So from the available 100 timeunits process A gets 45 and B 55 units. B has about 25% more runtime compared to A but it gained only 10% more

NICE	Weight	NICE	Weight	NICE	Weight	NICE	Weight
-20	88761	-10	9548	0	1024	10	110
-19	71755	-9	7620	1	820	11	87
-18	56483	-8	6100	2	655	12	70
-17	46273	-7	4904	3	526	13	56
-16	36291	-6	3906	4	423	14	45
-15	29154	-5	3121	5	335	15	36
-14	23254	-4	2501	6	272	16	29
-13	18705	-3	1991	7	215	17	23
-12	14949	-2	1586	8	172	18	18
-11	11916	-1	1277	9	137	19	15

Table 1: Linux Kernel 2.6.38 priority to weight table (variable prio_to_weight)

runtime through the priorityswitch.

To reach those relations, the weights need approximately the relation described in equation 1, where *prio* is the NICE level.

$$weight(prio) = 1.25 \cdot weight(prio + 1) \tag{1}$$

A weight value describes the frequeuency of this task running in relation to other tasks. [7] [3]

There is another table, which describes the inverse weights. Those are required for virtual runtime calculations in the fair scheduling class. To reduce the number of divisions, the inverse weights are precalculated and stored in an array. The inverse weight is 2^{32} divided by the weight, as seen in equation 2.

$$invweight(prio) = \frac{2^{32}}{weight(prio)} \tag{2}$$

Inverse weight describes the relative task sleep time between running on the CPU. This is again only relative to the other tasks, because the weight value does not reflect any real time. This characteristic of the inverse weight is used for virtual runtimes, which is independent of a task's weight. The virtual runtimes are explained later only for HTFS.

With the help of the weight values it is possible to calculate the runtime for an entity depending on the runtime of another entity. The ratio between the weights is equal to the ratio between runtimes. To calculate the runtime for an entity, we can transform it to equation 3, e_1 and e_2 are entities.

$$runtime(e_1) = runtime(e_2) \cdot \frac{weight(e_1)}{weight(e_2)} \tag{3}$$

2.6 Fairness

The fairness is an important characteristic of a scheduler. It guarantees a similar weight-independent runtime for all tasks. But to compare runtimes we need to include the runtime relation between differently weighted tasks. This leads to the equation 4 for the unfairness of a situation. P is the set of all running tasks. We need to divide through the weight of the task to get a relative runtime that is independent of weight. This also is a key idea for the usage of virtual runtimes as described later in the HTFS section.

$$unfairness = \max_{p_1 \in P, p_2 \in P} \left| \frac{runtime(p_1)}{weight(p_1)} - \frac{runtime(p_2)}{weight(p_2)} \right| \tag{4}$$

The maximal possible unfairness of a scheduler is the important metric to compare the fairness.

2.7 Hardware Awareness

There are not many hardware dependent operations in the scheduler besides the hardware aware load balancing, which I explained in the "Hardware Hierarchy" subsection (2.4). An example for another hardware aware operation is the taskswitch. This is also called context switch because of the many registers, which have to be stored/loaded. Also the virtual address space has to be adjusted for the new task to prevent a wrong address translation.

The context switch is implemented in the architecture specific code section of the kernel, the arch directory. The registers are different on several architectures, so most of the architectures have their own context switching routines. Through this seperation of architecture specific code and the scheduler, the scheduler does not have to deal with any hardware problems. The programmer only has to think about the correct sizes for the primitive datatypes because the C standard only declares lower size limits for the types. To get an exact size there are datatypes like "u64", which define for every architecture an unsigned integer of 64 bits. But this problem is also present in userspace programs. [3]

2.8 Taskgroup

Taskgroups add the possibility to group tasks and give them a share of the available runtime. The groups can include subgroups and tasks. Every taskgroup has a schedulable entity for each CPU. The weight of the schedulable entity is approximately calculated as shown in equation 5. se is the schedulable entity, tg the taskgroup, $TGRQS$ the set of all runqueues belonging to the taskgroup, the share function returns the share of a taskgroup defined through the user or kernel code.

$$weight(se) = share(tg) \cdot \frac{weight(rq(se))}{\sum_{rq \in TGRQS} weight(rq)} \tag{5}$$

It simply divides the taskgroup's share in pieces for the SEs dependent of the load of their runqueues. Through this calculation there is no problem with having a subset of all tasks in a taskgroup on different CPUs. All groups and tasks of this taskgroup are queued on one of the SE's runqueues. This leads to a tree structure for tasks. An example of a task tree is presented in Figure 1 included in the cfs rq. So a SE belongs to a taskgroup or a task. [3]

There are two ways to setup taskgroups. The first is to use a taskgroup filesystem and mount it somewhere in the Linux filesystem. Creating or removing a directory in this mount point is equivalent to creating or removing a taskgroup. By writing a process ID to the tasklist inside a taskgroup directory, the process is added to the group.

The other possibility is the kernel's automic taskgroup code. It automatically creates groups to increase the interactive behaviour of the system. This autogrouping patch was written by Mike Galbraith as "automated per tty task groups" [8].

3 Related Work

3.1 Completely Fair Scheduler – CFS

This is the scheduler of Linux 2.6.38. It was written by Ingo Molnar [9] and introduced in the kernel with the 2.6.23 release. CFS is very precise at the fairness. It uses virtual runtimes to decide which task is the next one to run. All tasks are stored in a red black tree with the virtual runtime as key. CFS is in the $\mathcal{O}(\log n)$ class because of a tree based design. CFS is a decentralized scheduler with one runqueue per CPU. This reduces the number of failing lock acquisitions and therefore reduces the runtime of the scheduler. The timeslices are variable and calculated to match certain latency requirements for all tasks.

CFS is intended to fit the workloads of servers and normal desktop PCs. However there were some complaints about bad interactivity support for this scheduler. After a time there was an autogroup patch released, which decreased the interactivity for desktop PCs while running heavy load in the background.

3.2 Brain Fuck Scheduler – BFS

Con Kolivas already programmed multiple schedulers for Linux, like the staircase scheduler. BFS is his newest scheduler [10] and still supported as a patch for the newest kernel versions, currently up to 3.0. BFS was released in 2009 and discards the classical Linux scheduling architecture. It is aimed at devices with only a few CPUs and is known as a scheduler, that increases the interactivity of a system. In some single core or dual core benchmarks the scheduler is even faster than CFS but it does not scale to systems with many CPUs, maybe because of the lock contention problem caused by using only a single runqueue.

3.3 Previous $\mathcal{O}(1)$-Scheduler

This scheduler was also designed by Ingo Molnar in 2002 [11]. The scheduler has a multiqueue design with one runqueue per CPU. Each runqueue has two priority ordered priority arrays which contain the tasks. One is declared as active, the other as expired. All tasks in the active array are waiting for their time on the CPU. After consuming their time, they are added to the expired array. If the active array is empty, both arrays are swapped. To increase the interactive behavior of the scheduler, the priority of long running tasks is decremented. Although the tasks priority is degraded, the priority differences between long running tasks are observed. The scheduler remains in the $\mathcal{O}(1)$ class through the array design. The scheduler is not fair as long as there are tasks which are leaving the CPU early. Also this scheduler did not support taskgroups.

3.4 Virtual-Time Round-Robin Proportional Share Scheduler

VTRR is the approach to get the concept of virtual time on the classical round robin scheduler. It was developed by Jason Nieh, Chris Vaill and Hua Zhong in 2001 and implemented for Linux [12]. VTRR can also be categorized as $\mathcal{O}(1)$. This scheduler is fair but the scheduler does not give early leaving tasks additional runtime. It uses a queue to order the tasks by their shares. For scheduling the queue is iterated and the next task is picked. The queue order only changes if the share changes or a task changes from or into the running state. In 2001, the Linux kernel did not have proper multicore or taskgroup support. This scheduler did not introduce those features.

3.5 PMQS: Scalable Linux Scheduling for High End Servers

PMQS is the Pooled Multi Queue Scheduler. The scheduler was developed in 2001 by Hubertus Franke et al. from IBM Thomas J. Watson Research Center [13]. This approach groups the available CPUs into pools where the balancing is done. This avoids balancing over all CPUs which would be expensive for high end servers with many CPUs. PMQS is an enhancement of the Multi Queue Scheduler proposed in “Enhancing Linux Scheduler Scalability” [14] which was developed by the same research group. MQS was developed for Linux 2.4 where only one runqueue existed for all CPUs. MQS proposes one runqueue for each CPU, which is the same as the 2.6 schedulers. Also, the pooling of CPUs is similar to the currently known architecture of scheduling domains where balancing is done in scheduling groups.

3.6 Enabling Scalability and Performance in Large Scale CMP Environment

This is a paper published in 2007 about a new scheduling architecture which is able to distinguish between Chip Multi-Processor (CMP) and Symmetric Multi Processor systems. The architecture was developed by Bratin Saha et al. from the Intel Corporation [15]. The main goal of this software is the efficient handling of CMP systems with more than 64 runnable hardware threads. So the software addresses three factors, fine-grained parallelism, programmability enhancements and heterogeneity. To achieve those goals, a userspace scheduling is used with a lightweight thread model. The software was developed for standalone usage to avoid limits from operating systems. The scheduling policies can be set from the userspace which is a really new concept for really high parallelity, and offers the programmer to better adjust the behavior for the special program.

4 High Throughput Fair Scheduler – HTFS

The name of the scheduler shows the main goal, which is high throughput while being fair to all tasks. For high throughputs we could reduce the time the scheduler needs for a scheduling request. Another possibility is to reduce the number of context switches because they are expensive. Beside the expensive switch of register contents, the cache is often not up to date for the next running task. This leads to more memory fetch requests than with a hot cache.

For a faster scheduler there are two important parts. The first is the independence of the number of processes, which are on this runqueue. So in short, it has to work within the boundaries of $\mathcal{O}(1)$. The $\mathcal{O}$ notation does not reflect the speed of the scheduler code, it only shows the dependence or independence of the number of processes. So another aspect is a high speed of the operations implemented by the scheduler.

For multicore systems the number of task migrations is another important factor because the cache of the target CPU most times is not populated with data for the migrated task. Also on some systems a task migration might need expensive data copying between different memory areas. So the next goal is to reduce unneeded task migrations.

Every scheduler has to handle interactive tasks in a way that shortens the runtime of the interactive tasks. The current Linux scheduler does not need any special behavior because of its design. So interactive tasks need to run fast.

To summarize the mentioned goals:

- $\mathcal{O}(1)$ operations
- Low constant time of the operations
- Fairness
- Reduce context switches
- Reduce task migrations
- Good interactivity

HTFS is embedded in the already described Linux kernel scheduling architecture as an alternative implementation of the fair scheduling class. I am using the kernel version 2.6.38.6 for the implementation. I am using a classical multiqueue scheduler design to reach the goals mentioned above. In total there are 100 queues per HTFS runqueue that are managed in a special way to guarantee fairness if all tasks are using their complete timeslices. All tasks using their complete timeslices is not realistic in modern systems, so there is a virtual runtime to correct the task's runtime by setting a boosted weight. Interavtive tasks are treated on a seperated runqueue to achieve small latencies for newly woken tasks or newly created tasks.

For better throughput I use dynamic timeslices. The number of task migrations is reduced by the introduction of an average load for tasks to calculate their real weight.

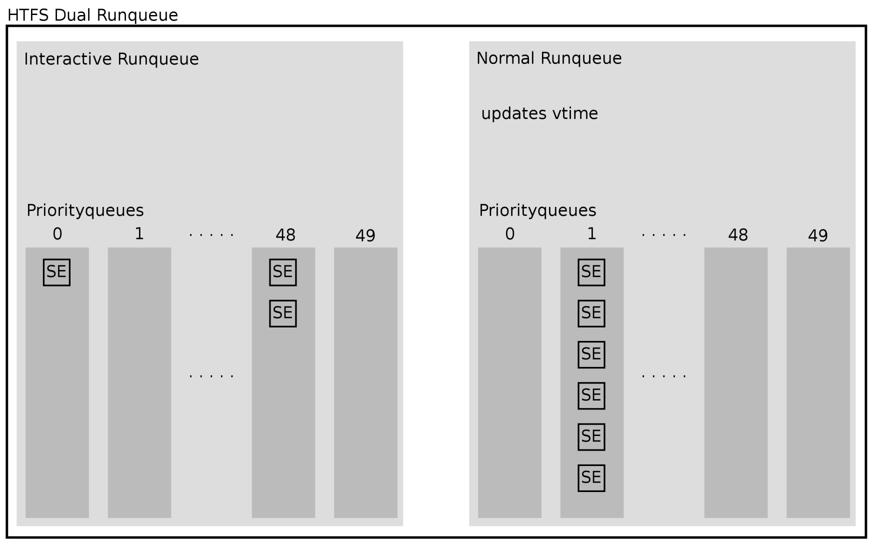


Figure 2: An overview over the HTFS specific runqueue structure named Dual Runqueue

Figure 2 presents an overview of the HTFS runqueue structure. A Dual Runqueue has two runqueue structures, one for interactive tasks, the other for normal tasks. The tasks are queued on one of the 50 priorityqueues each runqueue has.

4.1 Operation Sequences

At the beginning I will outline the three main operations a scheduler has to implement. To describe the HTFS behavior I will embedd the description in the general scheduler actions happening. I will not include taskgroup support in this sequence description because it only abstracts the behavior.

4.1.1 New Task

A new task might be a thread or a new process. In both cases the task is mainly initialized outside the scheduler code. Then the core scheduler starts to interact with the fair scheduling class. In multi core environments first the scheduling class has to find a good CPU where to start the task. Then the core scheduler acquires the necessary CPU locks and gives HTFS the task to enqueue on the HTFS internal runqueue. HTFS starts by intialzing some of the task's variables, for example the virtual runtime and boosted weight. The HTFS runqueue is the “HTFS Dual Runqueue”, which has two seperated runqueues inside. To enqueue a task to the dual runqueue, there are some state variables that has to be updated and the task has to be enqueued to one of the runqueues. To decide if a task should go on the interactive or the normal runqueue, the virtual runtime of the task is checked. Enqueueing the task to one of the runqueues is again only an update of some variables followed by picking the right priorityqueue and enqueue the task there. There are 50 priorityqueues matching to the 50 boosted weight levels, so the picking of the priorityqueue is easy. Enqueueing on a priorityqueue involves a normal list insertion at the end of the queue and updating of variables and possibly the position of the priorityqueue between the other ones. We might need to reorder the priorityqueues to be able to fastly update some essential variables while picking a task.

4.1.2 Task Done

When a task finishes, it is dequeued from the runqueue. It is mainly the same sequence as the reverse new task sequeuence. So the core scheduler calls the task dequeue function of HTFS. At this point it starts in the lowest structure, the priorityqueue, where the task is removed from the list and variables are updated. The runqueue dequeue and dual runqueue dequeue only include variable updates. HTFS does not have to care whether to put a new task on the CPU. This is all done by the core scheduler. Other parts of the kernel finally free the task structure and care about the correct actions to be done.

4.1.3 Schedule Request

Most times a schedule request is triggered by one of the scheduling classes. They are called every tick and decide if a reschedule is necessary due to the end of a timeslice or some preemption. First the core scheduler puts the previous running task back on the runqueue, then picks a new task and triggers makes a context switch if necessary. When putting the previous task back on the runqueue, HTFS decides if the task has to switch between the interactive and normal runqueue or between priorityqueues by checking the task's virtual runtime. This always includes some variable updates and is done like described in the above two subsections.

To pick the next task, HTFS decides from which runqueue a task should run on the CPU. This is done by using a balance-runcost model. With the same model the appropriate priotyqueue is chosen in the runqueue. The task in the first position on the priorityqueue is picked to run next.

After getting the next task, the core scheduler compares the previously running and the new task. If those tasks are different, the core scheduler executes a context switch, which replaces the register contents and other things on the CPU. Then the new task is able to run.

4.2 Priorityqueue

A priorityqueue (PQ) is a simple construct to store all schedulable entities with the same *bweight*. *bweight* is the boosted weight of a schedulable entity. It is similar to the normal weight of a SE. I will describe details and the calculation in the “Boosted Weight” section. Boosted Weight values are not continous like the normal weights. There are only 50 different *bweight* levels.

The schedulable entities are queued on the queue in the priorityqueue in first-in-first-out order. The queue is implemented as the standard doubly linked list of the kernel. Instead of a singly linked list, the doubly linked list allows a delete operation in constant time. The singly linked list would have $\mathcal{O}(n)$ time complexity. Additionally a priorityqueue has a sum of all *bweights* of the entities on the queue. This value is called *realbweight*, see equation 6. This valus is required to know how much runtime this PQ should get compared to other PQs.

$$realbweight(pq) = \sum_{se \in Queue} bweight(se) \tag{6}$$

4.3 Runqueue

The HTFS runqueue is not equivalent to the general runqueue. It is called runqueue because it can handle schedulable entities while obeying their weights. The runqueue has the purpose of maintaining 50 priorityqueues. The enqueue and dequeue operations of SEs on a runqueue are easy. There are only 50 discrete *bweight* levels, so a *bweight* index and the arrangement of the priorityqueues in one array makes it a simple array access. After finding the right priorityqueue, the *realbweight* is adjusted and the entity enqueued to the queue.

The operation of picking the next SE that is allowed to run on the CPU is more difficult. All priorityqueues have the *realbweight* value, which declares the weight of the PQ. So to simplify this, we have to divide the runtime into pieces for every entity while observing the weights. HTFS has adaptive timeslices. The timeslice is used for all tasks on the CPU, so the only possible way to reach different runtimes is to vary the frequency of running an entity from a PQ. Here I am using a model of run cost and a balance to effort a run. So every PQ has a balance counter, called *balancect*. The PQ is allowed to run exactly $\frac{balancect}{runcost}$ SEs. The *balancect* is increased in turns by the appropriate PQ. The following algorithm shows the idea of the algorithm:

1. Find a PQ with $balancect(pq) \geq runcost$
2. Run an SE from this PQ and $balancect(pq) = balancect(pq) - runcost$
3. If there is no PQ, increase all balancects: $balancect(pq) = balancect(pq) + realbweight(pq)$

The algorithm searches for any PQ with enough balance and runs a task from this. If there is no PQ left, all PQ's *balancects* are increased.

To be able to use this algorithm without problems like grouping same priority tasks, the *runcost* has to be so small to avoid that any PQ can run multiple tasks in one turn. At the same time it has to be big enough to run at least one task per turn. So the best *runcost* is the biggest *realbweight* value of all the priorityqueues because this PQ has to run a SE more often than any other PQ. This solves both problems that could occur.

To find the biggest *realbweight* the PQs are stored in a sorted list. This makes the enqueue and dequeue operations slightly more expensive because the PQ might have to be sorted to a new position in the sorted list. Of course this sorting keeps in the $\mathcal{O}(1)$ limitation through the constant number of priorityqueues. Also we only need to resort one element at a time, so the operation is really cheap and shifts the PQ most times by only one or two positions. The currently empty priorityqueues, those with a *realbweight* of 0, are not on this sorted list. This should save some more operations when a priorityqueue gets populated because the sorting algorithm does not have to compare with empty priorityqueues. Also the pick next task operation profits from the list of priorityqueues being sorted. It does not need to check all priorityqueues for their *balancect*, only the non empty ones.

Now we can look at the correctness of this algorithm. As I already mentioned in the "Basics" section, the weight values decribe relative to other weights, how much runtime an entity should get. I am assuming that the schedulable entities are not changing their *bweights*. Because the SEs are reduced to a limited number of *bweights*, this system can't be fair to them directly. For the fairness on SE-level there is the virtual time concept described later. So I am looking at the fairness between PQs at this point.

First I am defining the runfrequency of an entity. This is equivalent to the runtime. $runfreq$ is always the ratio of the biggest *realbweight* to the *realbweight* of the entity that is described. This is the same to the algorithm where *realbweight* is added to the *balancect* until the *balancect* is greater than the *runcost*. This gives us a run every $\frac{runcost}{realbweight}$ or a runfrequency as described in equation 7.

$$runfreq(e) = \frac{realbweight(e)}{runcost} \tag{7}$$

Starting with the runtime relation for weights from equation 3 I will show the equivalence to a form based on the runfreqeuncy.

$$\begin{aligned}
runtime(e_1) &= runtime(e_2) \cdot \frac{weight(e_1)}{weight(e_2)} && (8)\\
&= runtime(e_2) \cdot \frac{weight(e_1) \cdot runcost}{weight(e_2) \cdot runcost} && (9)\\
&= runtime(e_2) \cdot \frac{\frac{weight(e_1)}{runcost}}{\frac{weight(e_2)}{runcost}} && (10)\\
&= runtime(e_2) \cdot \frac{runfreq(e_1)}{runfreq(e_2)} && (11)
\end{aligned}$$

So the runtime relation is obeyed by the algorithm. Through normal round robin on the priority queues, all SEs are getting the appropriate runtimes depending of their *bweight*.

There are some implementation details to speed up the algorithm. Instead of first increasing all *balancects*, the implementation increases the *balancect* of a PQ short before checking the *balancect*. This saves a looping over all priorityqueues for increasing the *balancects*. Also the "pick next task" algorithm iterates over the sorted list instead of the array of PQs to ignore the empty ones. This also offers the possibility to jump over the conditions if the implementation reaches the end of the sorted list. This saves some pointer operations, an addition, an substraction and a comparison.

If a priorityqueue is getting empty and the currently running task is from this PQ, the pointer to the current priorityqueue has to be set to another PQ. This pointer is used to know, which priorityqueues are already checked by the algorithm for runnable tasks. The implementation tries to take the next priorityqueue and increases the *balancect* of that PQ. If there is no next priorityqueue, the previous priorityqueue is picked as the current one. Those conditions are required because the next element could also be the end of the list. This behavior saves a comparison for the *balancect* of the algorithm in most cases because we directly choose the next PQ whenever possible. Also the *runcost* is updated if the PQ was the one with the highest *realbweight* before.

The runqueue itself also has the sum of all *realbweights* stored as *bweight* for the Dual-Runqueue.

4.4 Dual-Runqueue

The Dual-Runqueue combines two HTFS runqueues to a full runqueue. The one is for interactive tasks only, the other is for normal tasks.

The seperation of interactive and non interactive tasks is important in this design to reach a good latency for interactive tasks. In the current implementation the weight of the interactive runqueue is 16 times higher than the *bweight*. The algorithm for picking the right runqueue for getting the next task is very similar to the one used for picking the priorityqueue. But here are only two runqueues so this is a simpler special case for the choosing algorithm, where no *balancect* on each runqueue is needed. Instead I am using a single *balancect* on the Dual-Runqueue. If this is below 0, the normal runqueue is picked, else the interactive runqueue. The *runcost* is replaced by the *bweight* of the other runqueue. So running a task from the interactive runqueue decreases the *balancect* by the *bweight* of the normal runqueue (equation 12).

The increment of the *balancect* when running an SE from the normal queue is more complicated. To give the interactive runqueue the 16 times more weight, we simply can define the *runcost* for the normal runqueue to be 16 times the *bweight* of the interactive runqueue. This is a multiplication so we don't drop any information at this point. If we would divide the *runcost* of the interactive runqueue we could loose some accuracy. Also the timeslices for normal runqueue and interactive runqueue are different. So this difference has to be in the calculation for the balancing to give both runqueues the appropriate runtime. In equation 13 you can see that the sizes of the slices are declared as shifts. Shift means the computational operation, which is the same as an exponent to 2. So the shift of 4 is the actually 16 times higher weight of the interactive runqueue. You can find more information about timeslices in the "Timeslice" section. *interq* is the interactive runqueue and *normalq* the normal runqueue.

$$balancect(interq) = balancect - bweight(normalq) \tag{12}$$

$$balancect(normalq) = balancect + bweight(interq) \cdot 2^{sliceshift - intersliceshift + 4} \tag{13}$$

There is no mechanism at the runqueues to avoid a pick on an empty runqueue. So we should not pick a task from an empty runqueue at the dual runqueue level. The implementation does not change the *balancect*, although the operation would not result in any change. But especially the calculation of the *balancect* for normal runqueue should be expensive enough to make it conditional. Additionally, whenever a runqueue gets empty, the *balancect* has to be set to -1 or 1 according to the empty runqueue. This does not need any condition but guarantees that there is no pick next task operation on the empty runqueue. Also whenever the first task is enqueued on the Dual-Runqueue, the *balancect* have to be set to the correct value, wherever the new task is enqueued.

Newly woken tasks are always enqueued on the interactive runqueue, to give them more runtime at the beginning. Interactive tasks should profit from this as they can process their data after a short time and then sleep again. However there is a limit of how long a task may stay on the interactive runqueue. A task is allowed to be 1ms ahead before switching to the normal runqueue. The decision is done through the comparison of virtual runtime and virtual time, which is described in the next subsection.

4.5 Virtual Time

Many CPU schedulers are fair as long as the tasks always take the full timeslice on the CPU. But there are many situations where the task leaves the CPU much earlier than expected. It is not possible to compare runtimes directly because we have to obeye the weight of the entities. Exactly for this purpose there is the concept of virtual runtimes wich are comparable.

The current Linux scheduler, CFS, uses those virtual runtimes to sort the entities on the runqueue. To pick the next task CFS can simply choose the entity with the lowest virtual runtime. This is the reason for the $\mathcal{O}(\log n)$ complexity of CFS. This complexity does not match the goals of HTFS, so there has to be a new idea for the comparison. HTFS still has the virtual runtime for each entity, but to avoid comparison between all entities there is a runqueue wide virtual time, which allows to rate the situation of an entity by a single comparison. I will further explain both parts in the following sections.

4.5.1 Virtual Runtime

The runtime relation between differently weighted entities defined in equation 3 includes a multiplication with the weight. To be able to compare two runtimes without the need of a weight value the runtime has to be multiplicated with the inverse weight, see equation 15. Because of the multiplication with 2^{32} in the inverse weight definition, we have to divide the result of a multiplication by 2^{32}. The Linux kernel is only able to store integers, so to save the result without loosing too much of the accuracy there is a *vtimeaccuracy*. This is a constant value such that the comparison of virtual runtimes is still possible.

$$vtimeaccuracy = 14 \tag{14}$$

$$vruntime(se) = \frac{runtime(se) \cdot invweight(se)}{2^{32-vtimeaccuracy}} \tag{15}$$

This general equation for the vruntime is perfect for steadily running tasks. The problem is that there are tasks that sleep between running. There are multiple possibilities how to deal with awaking tasks. The easiest way is to leave the *vruntime* unchanged for the task. But in a real system it is unreasonable to give a sleeping task that much additional runtime. Another solution is to set the *vruntime* to the minimal *vruntime* that is currently present on this runqueue. This could slow down interactive tasks as they do not get the needed time after sleeping to process the event they waited for. Also this solution is not really fair. So I decided for a half fair solution, to change the *vruntime* only if the *vruntime* is far too low or high compared to the virtual time. The threshold for the *vruntime* is defined in equation 18. There is a statictimeslice in the equation because of the variable timeslices, as described later. So the vthreshold describes that the entity's *vruntime* is allowed to differ from the virtual time up to 2^6 statictimeslices.

$$statictimeslice = 2^{17} ns \tag{16}$$

$$wakeupthr = 10 \tag{17}$$

$$vthreshold = \frac{invweight(se) \cdot statictimeslice}{2^{32-wakeupthr-vtimeaccuracy}} \tag{18}$$

Another modification of the normal virtual time model is the punishment of early leaving tasks. The computer is no ideal system, so context switches take time. So to represent this thought in the virtual runtime, early leaving tasks get a punishment in form of an additional virtual runtime increase. The *vruntime* is increased by one eighth of the granted timeslice. Of course there is a runtime tolerance of 2^{14} to avoid punishment of tasks, which used nearly the complete timeslice. Preempted tasks do not get any punishment.

It is important for virtual runtimes to implement everything safe against overflows, which will occur after some time. A limited number system has the nice characteristic of calculating correct results within this system for all operations. For example for an 8 bit unsigned integer this equation is correct: $255 + 1 = 0$. Let us assume $vtime = 3$ and a virtual runtime of some task is $vruntime = 250$. A simple comparison of *vruntime* and *vtime* will give us a wrong image of the situation, that the *vruntime* is ahead of the *vtime* by 247. In fact if we calculate the difference of both in this number system, we get: $vruntime - vtime = 250 - 3 = 247$. But it is a difference in the 8bit number system. Differences are always signed numbers. So if we cast the result to a 8bit signed integer we get the result of -9. Now we know that the *vruntime* is 9 units behind the *vtime*. This example shows that direct comparisons in a limited number system are dangerous, but signed differences give us the real result. This is what I did in the implementation to be able to handle overflows.

4.5.2 Virtual Time

The runqueue-wide virtual time is equivalent to the *vruntime* of a perfectly running task, without being preempted or leaving the CPU too early. This virtual time allows to compare the non ideal virtual runtimes of the tasks with this ideal time. Virtual Time offers the possibility to determine whether a task needs more or less time on the CPU with the effort of only two comparisons. The first to check for a positive or negative difference, the second to check if the difference is within a threshold. The actual analysis of the comparison is described in the next section.

The interactive runqueue gives the entities additional runtime compared to the normal runqueue, so it is enough to update the virtual runtime of an ideal running task whenever a task of the normal runqueue is running. We can reduce the complexity of when to increase the virtual runtime by taking the complete normal runqueue as a running entity. The weight of the normal runqueue is the sum of all *realbweight* values of the priorityqueues. The assumption of the normal runqueue being a running entity gives us the possibility to update the *vtime* after every time a task from the normal runqueue ran.

$$vtime = vtime + \frac{time \cdot \frac{2^{32}}{realbweight(normalq)}}{2^{32-vtimeaccuracy}} \tag{19}$$

Equation 19 shows the virtual time modification with *normalq* being the normal runqueue. If the previously running task was preempted, *time* is the runtime of the task, else *time* is equal to the granted timeslice. We need to differentiate between those cases because tasks does not get punished when they are preempted. This can lead to *vtime* being increased much faster than the tasks getting runtime, which results in massive boosting of the tasks.

4.6 Boosted Weight

Through the virtual time and runtimes we know if a schedulable entity does need more or less runtime. To implement this, I decided to introduce a boosted weight that is adjusted dependent of the virtual runtime.

The boosted weight increases or decreases the frequency of the task getting time on the CPU, what directly influences the task's runtime. It is more expensive to change the boosted weight than just reenqueue the entity to the previous queue. So we have to reduce those boosted weight changes for a higher scheduler speed. Also there is an big overhead for continous boosted weight values because we can only effort a constant number of queues to stay withing the $\mathcal{O}(1)$ limitations.

HTFS uses an adaptive approach, so the boosted weight should reach a stable level after a time. Boosted weight has 50 discrete levels. The 50 values consist of the 40 weights that are defined for the priorities and another 5 values on each end of the range to be able to compensate too much or less runtime.

To initialize the *bweight* of an entity, a binary search is used to find the index of the best matching *bweight*. Through the nature of binary search, the first element that is checked is the standard priority, NICE level 0. This saves some loop passes for the majority of entities. The found *bweight* for the schedulable entity is increased by one. It is getting this little boost at the beginning to lower the latency of new entities.

After running on the CPU the schedulable entity's *vruntime* is checked if the *bweight* has to be modified. There are multiple conditions before the *bweight* is actually changed. First the *badness* has to be unequal to 0. *badness* reflects if the absolute difference between *vruntime* and *vtime* in timeslices is above the boost threshold. See equation 22.

$$vdiff(se) = vruntime(se) - vtime \tag{20}$$

$$boostthr = 2 \tag{21}$$

$$badness(se) = \frac{|vdiff(se)| \cdot weight(se)}{slice \cdot 2^{vtimeaccuracy+boostthr}} \tag{22}$$

Every schedulable entity stores the last calculated $vdiff$, which is used the next time to know if the difference between *vruntime* and *vtime* is still growing. If this is the case, the *bweight* is increased or decreased by one step. Else the last *bweight* adjustment seems to be enough and the *bweight* is not changed. The usage of a last $vdiff$ prevents the *bweight* and the *vruntime* of the schedulable entity from oscilating. In the end the *bweight* will toggle between two *bweight* levels, the best matching ones. So after a time the *bweight* reaches a stable value for every schedulable entity and workload. A problem might be tasks that have changing workloads.

After every *bweight* change, the task has to be dequeued from his old priorityqueue and enqueued to a new one. This triggers two times a resorting of an element in the sorted priorityqueue list together with a possible removal from the list and a new division of the inverse *realbweight* for the *vtime* increasement.

The boosted weight system is able to guarantee fairness. The *bweight* values are limited, so we have two critical parts to look at. First if the *bweight* decreases this means the entity has too much virtual runtime. This can only happen if the *bweight* was higher than the normal weight, or because of a small inaccuracy at the timeslice end. Both cases are correctable through one or two *bweight* drops. So the *bweight* will never reach the end of range. Also the minimal share of a taskgroup is 2. There is another *bweight* level below dropping the frequency for this entity to a half. The second case, if a task has too less virtual runtime, there are some *bweight* levels to compensate this. If this is not sufficient and the difference between *vruntime* and *vtime* continues to grow, the entity is enqueued to the interactive runqueue. I already mentioned the interactive runqueue is boosted 16 times. Together with the punishment for early leaving tasks, which is one eighth of the granted timeslice, the difference becomes smaller. All conditions for this system are independent of the number of processes. They are dependent on timeslice sizes, but timeslices are limited in both directions. This leads to the conclusion that boosted weights in HTFS limit the possible unfairness to a constant maximum. So this scheduler reaches the $\mathcal{O}(1)$ class of unfairness. However in most cases the fairness of CFS should be much better than HTFS because of the strict checking of *vruntimes* in CFS.

Entities, which wake up from sleeping might have changed their workload, for example through thread pools in the application. But the wakeup could also be normal through some waiting and belong to the workload of the task. So I decided to not change the boosted weight. The task from a thread pool might need a little longer before reaching a stable boosted weight, but the other tasks might profit from this solution. So the *bweight* is kept after sleeping.

In the "Dual-Runqueue" section I described the interactive runqueue to get much more runtime than the normal one. Of course this reflects in the *vruntime* of the schedulable entities in comparison to the virtual time. So the expected behaviour for long running tasks, which start running is to stay on the interactive runqueue for several timeslices. After reaching the threshold it is enqueued to the normal runqueue. Because of the small boost for new entities, it will reach a $vdiff$ above the threshold where the *bweight* is decreased until the task does not increase the $vdiff$ anymore. From this moment on the task should run relatively stable with the *bweight*.

4.7 Taskgroup Support

The taskgroup support of HTFS shares most of the parts with the CFS scheduler. There are some modifications to the other concepts of HTFS to achieve better behaviour for taskgroups.

The behavior of the "pick next task" changes for taskgroups. The function begins at the root taskgroup and repeatedly calls the pick next task for the current runqueue. This ends with reaching a leaf schedulable entity. HTFS switches the behavior as soon as the chosen entity is on the interactive runqueue. From this moment on HTFS prefers to choose interactive entities. HTFS should increase the system's interactivity on taskgroup configurations this way.

To better use the above described picking mechanism the enqueue and dequeue operations are modified. Task enqueueing begins with the leaf schedulable entity and goes through the group hierachy path to the root. HTFS does nothing new while enqueueing entities, which are not on the runqueue. As soon as the first already enqueued entity is found, HTFS tries to pull this entity on the interactive runqueue if all already enqueued entities are on the interactive runqueue and the *vruntime* of the entity does allow the move to the interactive runqueue. This is done for all following SEs until any SE could not be moved to the interactive runqueue or the root taskgroup is reached. Through the pick behavior and this enqueue function, the probability of running newly woken tasks after a very short time is relatively high.

For dequeueing of tasks, HTFS tries the opposite. The operation also starts from the leaf entity. If the runqueue the entity was deleted from does not hold any further entities, the entity holding this runqueue is also deleted. If the interactive runqueue is empty, HTFS tries to move all above SEs to the normal runqueue, as long as the interactive runqueue is empty. This removes entities that have no reason to be on the interactive runqueue anymore.

4.8 Task Preemption

This feature reduces the latency for tasks waking up by preempting the currently running task from the CPU. Obviously there have to be some conditions where the task is not preempted. HTFS does not preempt any interactive task from the CPU or any task that has less time to run than an interactive timeslice. The wait time for the newly woken task is very short if only this interactive timeslice remains. Also the task is not preempted in case a scheduling is already requested. To remain fair, the preempted task does not get a punishment for leaving the CPU early.

In contrast to CFS, HTFS can not guarantee that the task is the next one on the CPU. A preemption only results in running one or more interactive tasks. That could lead to higher latency, especially when running a setup with multiple taskgroups.

4.9 Timeslices

HTFS has static timeslices. There is a global interactive timeslice of $2^{17}ns$ and a CPU-wide normal timeslice. The normal one is changing over time to match the current workload better on this CPU. For workload independent calculations there is a static timeslice, which is never used as a timeslice for a task.

All the available time is divided into normal timeslice sizes. A normal timeslice can be used by one task or can be divided into interactive timeslices. As soon as there is less time remaining than a normal timeslice the scheduler tries to only run interactive tasks. If there is no interactive task remaining, a normal task gets the remaining time. This behaviour is implemented to handle a group with interactive entities, which is enqueued on the normal runqueue of another group. If the complete group gets a complete timeslice to run, this mechanism assures that the running of an interactive task will result in running more interactive tasks.

The task that is next to run gets an interactive timeslice if any of the SEs in the hierarchy above this task is on the interactive runqueue. If all entities are on the normal runqueue, the task gets the full timeslice.

4.9.1 Adaptive Timeslices

To reduce the number of contextswitches for long running CPU bound tasks, the normal timeslice is modified after a time. This is done by analyzing the waiting time of all entities on the interactive runqueue. As soon as a task waited for more than 10ms an indicator is increased, which is called *sliceshrink*. The indicator is checked frequently. As soon as the value is greater than 1, the normal timeslice is divided by 2 and the indicator is reset to 0. After $10ms$ the indicator is set to 0. If the indicator is 0 after $100ms$, the size of the timeslice is doubled again. The maximum timeslice is $2^{30}ns = 1.07s$, the minimum is equal to the interactive timeslice of $2^{17}ns$.

Event	**sliceshrink**	**Timeslice**
init	0	2^{30}
latency too high	1	2^{30}
latency too high	0	2^{29}
latency too high	1	2^{29}
10ms	0	2^{29}
latency too high	1	2^{29}
10ms	0	2^{29}
90ms	0	2^{30}

Table 2: Example sequence for the adaptive timeslice

Table 2 shows an example sequence for the adaptive timeslices. The different events are “latency too high”, triggered by an interactive task waiting to run for more than $10ms$, and “10ms”, triggered after 10ms.

This system should slowly adapt to new workloads like fully interactive or fully CPU bound. Whatever workload is currently on the CPU, this tries to reach a maximal latency for interactive tasks of 10ms. The decrease of the normal timeslice does give the interactive runqueue earlier the possibility to run a task, which really decreases latency.

4.10 Average Load

Average loads are metrics for better balancing decisions. It is called average load but actually the average load is implemented as an estimation of the average which does not need any divisions. I will first describe the equation for the average and then later the changes for balancing.

The average is a very simple form of a moving average where the history is replaced by the value itself, see equation 23. It is a very inaccurate average but reduces the calculation by some operations. When choosing $k = 2^x$ the division can be replaced by a shift operation. The kernel only offers integer operations, so for a better rounding of the value, we have to add 0.5 because all integer operations round to the lower full value. Equation 24 shows how the 0.5 is integrated into the equation. A simple addition of 0.5 at the end would not solve the problem, because the operation would have already removed the necessary information for correct rounding. Also the representation of 0.5 as integer is not possible.

$$averageload = \frac{averageload \cdot (k-1) + newvalue}{k} \tag{23}$$

$$averageload = \frac{averageload \cdot (k-1) + newvalue + \frac{k}{2}}{k} \tag{24}$$

After an entity slept the average has to be updated. In this case the *newvalue* is 0 because the entity did not run and the produced no load. We know how often the calculation would have been triggered while sleeping. So we can create a calculation, which calculates the new average in equation 25. Instead of using a multiplication this can be transformed to a division. The divisor is then a value greater than 0. HTFS always uses $k = 2^3$ so we can try to find an exponent for the fraction, which is near to 2, as seen in 27. We can not use the exponent 5, that would lead to a division of turns by 5. So we have 4 instead as an estimation, which gives us the possibility to divide by 4, or use the shift operation. This directly leads to an estimation that is cheap to calculate, equation 28.

$$averageload = averageload \cdot \left(\frac{k-1}{k}\right)^{turns} \tag{25}$$

$$averageload = \frac{averageload}{\left(\frac{k}{k-1}\right)^{turns}} \tag{26}$$

$$\left(\frac{8}{7}\right)^{4} = 1.706 \tag{27}$$

$$averageload = \frac{averageload}{2^{\frac{turns}{4}}} \tag{28}$$

Every schedulable entity and every dual runqueue has an average load. For the schedulable entity the average load is initialized to half of the weight. It is updated after each time the entity was on the CPU. The average is updated with the weight of the process. If the entity gets punished, this punishment is accounted as unused time. The average load is then updated with $weight(se) \cdot \frac{timeslice-unused}{timeslice}$. It simply expresses the weight this task would have if the task always uses this runtime.

For the runqueue average load the sum of all average loads of schedulable entities are the *newvalue*. This guarantees only to be able to migrate a maximum of the really existing *bweights*. Also the cpu load is set to the runqueue average. The runqueue average is updated everytime after a task ran on the CPU.

All balancing algorithms are done by the CFS code. I only changed the task picking for migration because it uses scheduler specific runqueue structures. The CFS code decides, which task to migrate to another CPU by looking at the weight of the schedulable entities. I changed this to the average load. This should avoid often calling migrations because we already know the real weight of a task.

4.11 Lock Acquire Retry

Kernel internal locks are often implemented as spinning lock. So a task trying to acquire a spinning lock, repeatedly calls the lock function. To give other tasks the possibility to run while the other task cannot get the spinning lock, a schedule is called. It is directly called by the waiting task. With Multi-CPU systems it might be cheaper trying again to get the lock than rescheduling. CFS has such an mechanism through the design. As long as a task's virtual runtime is less than all others the probability of choosing the same task to run next is very high.

HTFS is by design not able to automatically pick the same task for the next timeslice. So I implemented a manual check if the task might hang at acquiring a lock. For this purpose the time of the schedule call and the last runtime between those calls is stored. Equation 29 shows the calculation of $rundiff$. If $rundiff$ is less than 2^5 a new task is scheduled.

$$rundiff = |lastruntime - runtime| \tag{29}$$

There is another condition, which checks for a very small runtime. So if the *runtime* is less than 2^8, there is also a new scheduling. To prevent this mechanism to fill a complete timeslice without doing anything but with varying runtimes, there is a maximum of continuations of 20. Also a timeslice that is nearly completely consumed will trigger a normal schedule. To avoid those checks at schedule calls, which are directly triggered by scheduler code, there is a "force schedule" flag. As the name says, it immediately forces a scheduling. The task choosen for continuing running is stored for the next task request of the core scheduler code. The enqueueing of the task is omitted as long as there is no real scheduling.

In the best case, this mechanism will give tasks on other CPUs the opportunity to release the lock while a task on this CPU is trying to acquire this without a context switch. In the worst case it takes this mechanism 2 or 3 schedule calls before making a real scheduling. If the detection does not work at all this will cost 20 schedule calls.

5 Evaluation

5.1 Test Environment

I am using the phoronix test-suite version 3.4 milestone 3 [16]. The test-suite is designed to automatically run tests. While running tests the tests-suite can log different system monitors. To better understand the test results I wrote some new system monitors, which log for example the context switches per second or the balancing operations per second. Together with the existing ones (first two items) I monitored the following values:

CPU usage records the usage of the CPU in percent of all CPU power available in the system.

IOWait monitors the percent of the CPU usage that is spent waiting for IO.

BalancingOps: Number of all balancing operations on the system per second.

Migrations: Task migrations between CPUs per second.

Active BalancingOps: Number of active balancing operations per second.

Active Migrations: Number of active task migrations between CPUs per second.

Context switches: Number of Context switches on the system per second. This is not the same as schedule requeuest because of the possibly further running tasks.

Created Processes: Number of new processes per second.

Running Processes: Number of currently running processes, measured once in a second.

Latency: Latency between a wakeup of a task and the first running. This is done by measureing the time of sleeping for 0.25 seconds. The time of inaccuracy is the latency in microseconds.

The monitors are called once in a second. To calculate the values for the monitors, I am using files in the Linux kernel filesystem. The raw data is most times a cumulative counter. So the result is always a difference of the new value and the previous one. Through the high load of the system while running some of the tests, it is not guaranteed that the calls for the monitors are really accurately once in a second. Unfortunately the test-suite is not able to record the time when a monitor returns the result. So for every value I am tracking the current time and recalculate the difference to a per second value. Unfortunately sometimes the monitors returned negative values. It might be an overflow of the cumulative counter or the possibly interrupted invocation of a monitor. However this did not happen very often. In the monitor graphs the lines are plotted for all testruns. So the graphs also include the time between test runs. There is no average or standard errors plotted in the graph. Instead you can find in the top left the average as a number together with a legend.

The phoronix test-suite already offers many tests to benchmark a system. But there are some standard benchmarks for schedulers that were not available for the test-suite. So these are the additional test profiles I wrote for the evaluation, which are using the already existing benchmarks, Table 3.

Test	**Parameters**	**Description**
Hackbench [1]	10,50,100,200,300 Processes	Hackbench creates a number of processes. Each process sends every other process a message. The performance is measured in seconds.
Sysbench	num-threads=32 test=cpu cpu-max-prime=10000	Sysbench CPU test measures the time needed for calculating prime numbers until a given number.
Sysbench	num-threads=32 test=threads thread-yields=10000	Sysbench Thread test calls the scheduler for by the sched_yield system call. The unit is seconds.
Sysbench	num-threads=32 test=mutex	Sysbench Mutex test creates a number of mutexes and tries to lock them and unlock while running multiple threads.
Sysbench fairness	num-threads=32 test=cpu cpu-max-prime=10000	This test uses sysbench to find out the maximal difference between the work done by different threads. So the result is the fairness achieved at this test. It is measured in standard deviation from average.
byte-unixbench	execl	The Unix BYTE benchmark "execl" test measures the speed of the "execl" system call. The unit is LPS, loops per second.
byte-unixbench	pipe	The Unix BYTE benchmark pipe test measures the possible pipeline throughput. Measurement unit is LPS, loops per second.
byte-unixbench	spawn	The Unix BYTE benchmark spawn test measures the possible process creations. Measurement unit is LPS, loops per second.
kernel build		Builds the Linux kernel version 3.0.4 with 2 times the number of CPU cores jobs. It measures the time needed for compiling in seconds.

Table 3: Created test profiles for the evaluation.

The following used test profiles are already defined in the phoronix test-suite, Table 4.

[1] Craig Thomas's hackbench: http://devresources.linuxfoundation.org/craiger/hackbench/src/hackbench.c

Test	Parameters	Description
Apache Benchmark	-n 700000 -c 100	Apache Benchmark measures the possible requests per second with 700000 requests and 100 concurrent threads. The served website is static html.
Bork File Encrypter		The Bork File Encrypter test encrypts a 2 GiB file. Bork is a java based crypt tool. For long term archiving, the encrypting software is included in the file.
Apache build		Builds apache version 2.2.17.
ImageMagick build		Builds ImageMagick version 6.6.3-4
MPlayer build		Builds MPlayer version 1.0-rc3
PHP build		Builds PHP version 5.2.9
CLOMP		C version of the Livermore OpenMP benchmark to measure OpenMP performance. OpenMP is a multi-thread library for easy parallel programming. This test measures the performance in speedup related to the singlethreaded execution.
7-Zip		7-Zip compression performance measured in seconds.
Gzip		Gzip compression performance measured in seconds.
LZMA		LZMA compression performance measured in MIPS.
Parallel BZIP2		Parallel BZIP2 compression performance measured in seconds.
FFmpeg		FFmpeg encoding performance test measured in seconds.
x264		x264 encoding test measured in FPS, Frames Per Second.
C-Ray		C-Ray is a raytraceing program to measure the floating-point CPU performance. This test runs with 16 threads per CPU.
Fhourstones		Fhourstones is an integer benchmark measured in Kpos/s, Kilo positions per second.
NGINX		This benchmark uses the apache benchmark program to measure the performance of a NGINX http server. Measurement unit is requeuests per second. There are 500000 requeuests with 100 threads running.

Table 4: Used test profiles from the phoronix test suite.

Every test ran at least 10 times. After the tenth run the test-suite calculated the standard deviation and repeated the test as long as necessary to get the standard deviation below 0.1%. The maximal number of runs per test was 20, which was most times reached because of the very low allowed standard deviation. All results have the standard error noted as “SE”.

All the above tests where executed with the AMD Cool’n’Quiet and Intel SpeedStep technology being disabled to reduce the number of factors that can influence the results. The test kernels were configured exactly the same concerning the scheduler preferences. High resolution timer was enabled, tickless system and autogrouping were disabled.

5.1.1 Test-Systems

To test HTFS on both big machine categories, I chose a desktop grade machine with a dual core processor and a server grade machine with two quad core processors using symmetric multi processing and hyperthreading.

Dual-Core System

Processor AMD Athlon II M320 @ 2.10 GHz

Motherboard LENOVO Bali

Memory 4096MB

Disk Mushkin Enhanced 60GB SSD SATA II

Operating System Gentoo Linux stable

This system was used for the single and dual core tests. Single core tests was done by disabling ACPI, which only activates one CPU.

Hexa-Core (Hyperthreading) System

Processor 2 x Intel Xeon E5520 @ 2.26GHz

Motherboard Intel S5520SC

Memory 12288MB

Disk Western Digital RE3 500GB SATA II

Operating System CentOS 5.6

Through the hyperthreading technology the system has 16 CPUs. Unfortunately on this system not all tests were able to run through deprecated packages of the operating system.

5.2 Single CPU results

The results presented in this section are only single CPU results to look at the performance of HTFS and CFS without the influence of the balancing methods. With single CPU systems, the balancing methods are not executed what clearly will show the performance of the scheduler on a single runqueue.

5.2.1 Scheduling Speed

First I want to discuss the speed of the HTFS scheduling code. Fast scheduling code can increase the performance of many programs. Of course the second part for the performance is the scheduling decision. To measure the speed we need a synthetic benchmark that often triggers a scheduling. For this I am using the sysbench threads test, which repeatedly calls “sched_yield”. This system call directly triggers a scheduling class call of the function “sched_yield” followed by a normal schedule request.

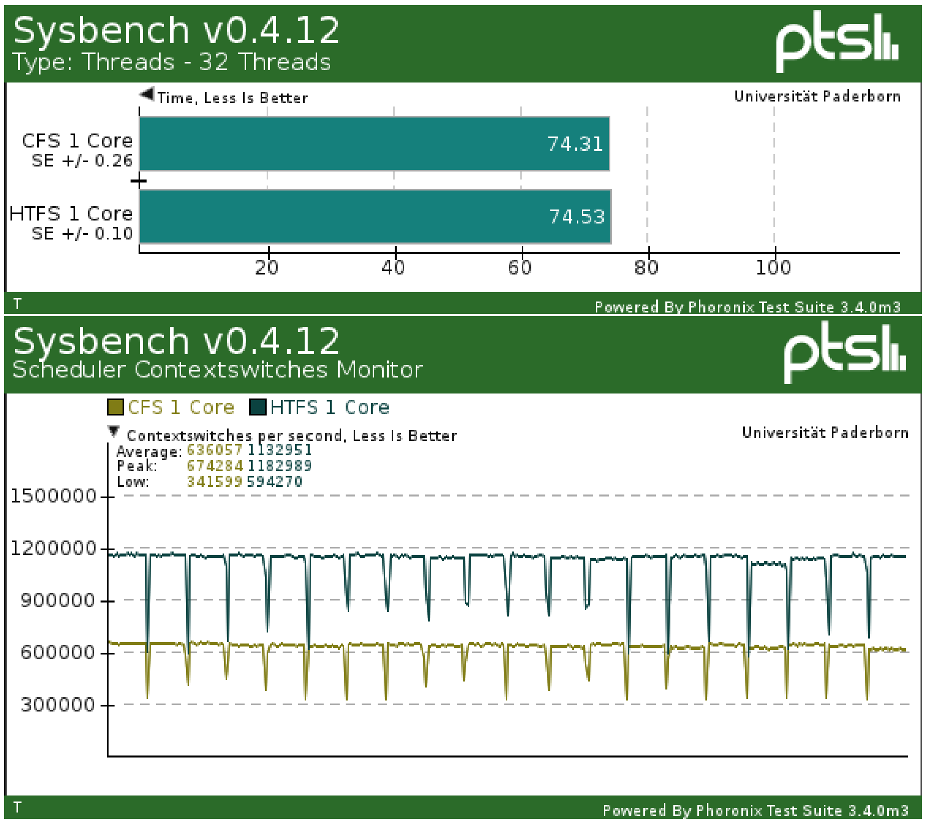


Figure 3: Sysbench threads single core performance and context switches.

Figure 3 shows the speed of the HTFS compared to CFS. HTFS is nearly as fast as CFS in finishing the test. But there is a big difference in scheduling speed. There are two factors contributing to the result. The first factor is the number of contextswitches done. This can vary because the task does not have to be switched after a schedule request. The second factor is the speed of the scheduling code.

So to find out how fast the scheduling code is, we have also to look at the number of contextswitches. HTFS has about 1.1 million, CFS only 0.6 million contextswitches. So HTFS spents significantly more time with context switching and still reaches about the same result like CFS, so HTFS scheduling speed is higher. The reason for the higher number of contextswitches is the correct detection of repeated schedule requeuests through the "Lock Acquire Retry" mechanism. For CFS only the virtual runtime of the task is of importance, which leads to a non flexible reaction on processes that really wish to leave the CPU. This behavior can decrease the performance of applications, which use "sched_yield" to leave the CPU. HTFS tries to seperate those tasks from the tasks, which implement locking mechanisms.

5.2.2 Latency

To look at the latency of HTFS the design suggests that there is a best case, when most of the tasks are on the normal runqueue, and a worst case, when all tasks are on the interactive runqueue. I will start with the worst case. To reach the worst case with the HTFS design is difficult. So I picked the worst measured latency for HTFS in all the tests. Hackbench with 300 processes fullfills the assumption of all processes being on the interactive runqueue, at least at the beginning of the test.

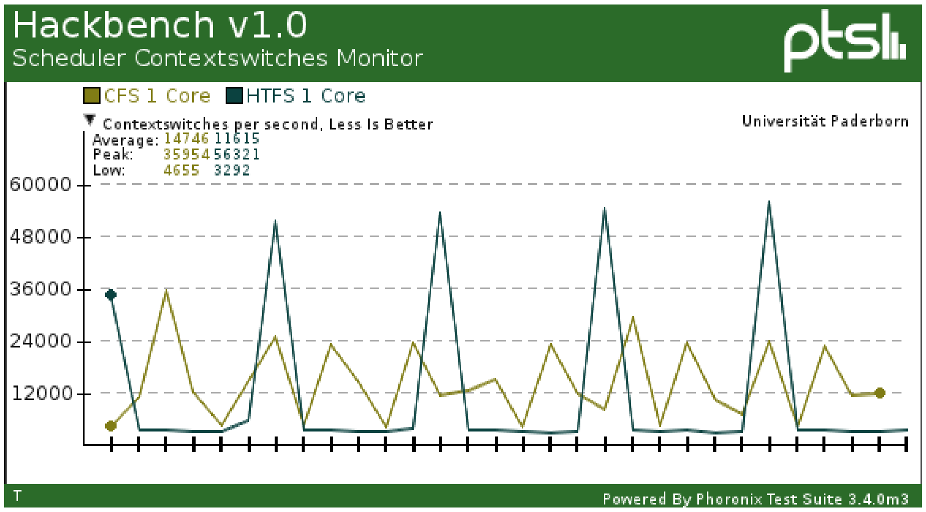


Figure 4: Context switches at hackbench with 300 processes.

Figure 4 shows those high peaks of contextswitches at the beginning of each testrun. The contextswitches of HTFS go up to about 50000 per second. This is the normal behavior after starting 300 processes for a test. All tasks are enqueued on the interactive runqueue, which causes a high number of contextswitches. You can see the high influence of 300 processes on the interactive runqueue in Figure 5. At the beginning of each testrun there is a very high latency peak between 40 and 100ms. After a short time the 300 processes are switching to the normal runqueue. You can see a direct drop in the latency after the peak, which is caused by the 300 processes switching to the normal runqueue. The worst peak for HTFS is 104ms. This also is the worst measurement in all tests.

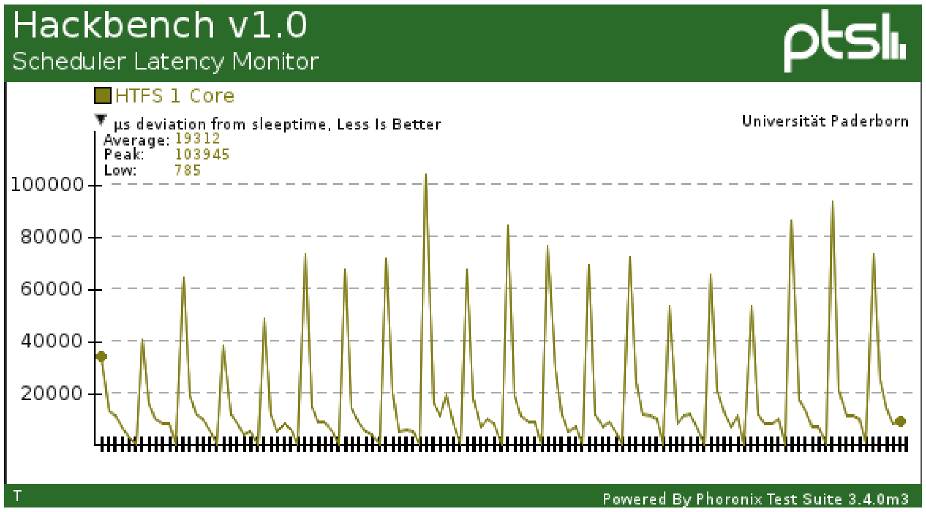


Figure 5: HTFS latency at hackbench with 300 processes

In Figure 6 you can see the latency of both schedulers. CFS's latency is much higher in Hackbench with 300 processes with a peak of 12 seconds and an average of 4 seconds. So this seems to be the worst case for CFS, too. With an average of 4 seconds, the system would be unusable for real users. So HTFS has better latencies than CFS even in the worst case. CFS is not able to distinct between different types of tasks. CFS only tries to always pick the task with the lowest virtual runtimes, which can produce really high latencies for some tasks. HTFS also has problems, but through smaller timeslices for interactive tasks it can achieve a better latency even with many processes active.

Figure 7 shows the normal situation of latencies for both schedulers. This also is the best case. Most of the time the awaking task is immediately getting time on the CPU. Only sometimes there is a small delay of about one or two additional milliseconds. But those short latencies do not influence any non realtime program.

Overall HTFS is able to offer better latencies for awaking tasks than CFS.

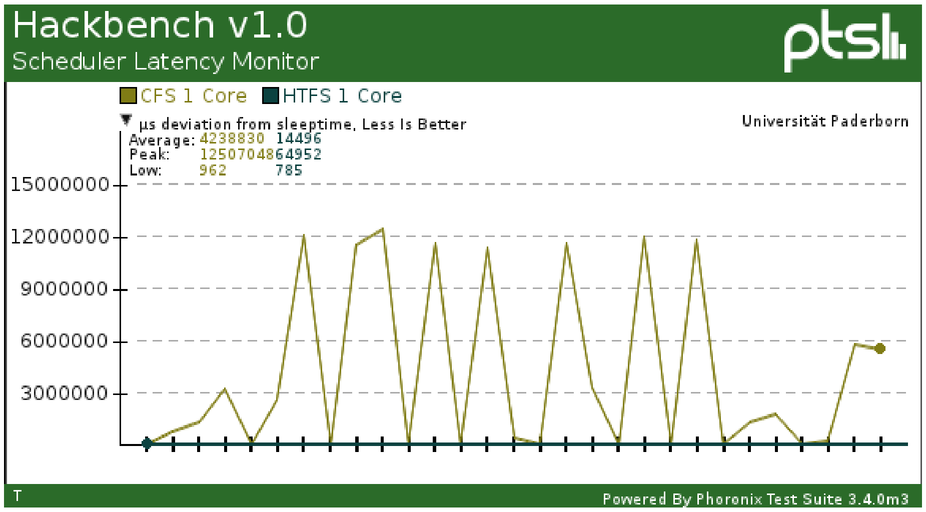


Figure 6: Latency at hackbench with 300 processes

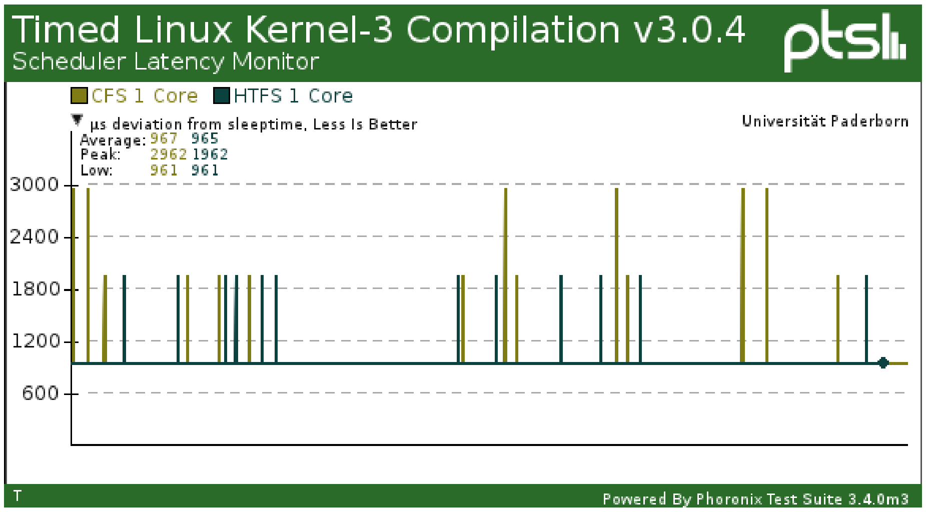


Figure 7: Latency at Kernel Compilation.

There might be some situations where the HTFS latency is worse than CFS's because of wrong categorized tasks as normal. But the tests used for this evaluation did not find any such problems. To reduce the latency problems with CFS there was a patch published for automated taskgroups [8]. By grouping different tasks started from different locations, like terminal or child processes, the patch can solve the problem of CFS for user interface programs.

5.2.3 Fairness

It is very difficult to measure the fairness of a scheduler. If the test program has a number of same processes, which measure the work done in a given amount of time, the only thing we will measure is possibly the length of the timeslice. All tasks will be scheduled the same way because they have the exactly same workload. If we use a test with different workloads, we have a problem in comparing the amount of work by each different workload.

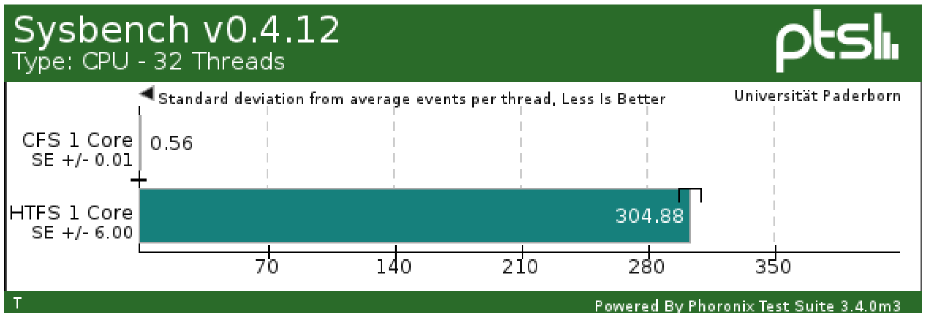


Figure 8: Deviation from the average work done per thread.

However the difference between CFS and HTFS was still visible in the standard deviation from the average work done per thread. Figure 8 shows the deviation from the average work done per thread in the sysbench cpu test. As you can see CFS is very strict with the fairness. A deviation of 0.56 is really good. HTFS obviously allows much higher deviations. Through the HTFS design it is simply not able to achieve such high accuracy. Sysbench results are always measured in loops per second or in this case the overall loop passes a thread has more or less than the average. So a standard deviation of 392 loop passes should not influence any application. A user could not sense this difference.

5.2.4 Throughput

There are a lot of different workloads we have to look at to get an impression of the throughput reachable with HTFS in comparison to CFS. I will begin with some synthetic benchmarks and continue with real benchmarks like file compression, compilation, website serving and video encoding.

Hackbench creates a number of processes, which communicate over sockets. Figure 9 shows the results for two tests. Both graphs show that HTFS and CFS need similar time to finish the test. HTFS is slower than CFS in the 100 processes test, while it gains the lead with 300 processes, so apparently HTFS scales better than CFS. I summarized the results for all Hackbench tests in Table 5 to create an overview of the speed in relation to the number of processes.

The worst result in this test for HTFS is the testrun with 100 processes. Below and above the per process time consumption is lower. CFS gets worse

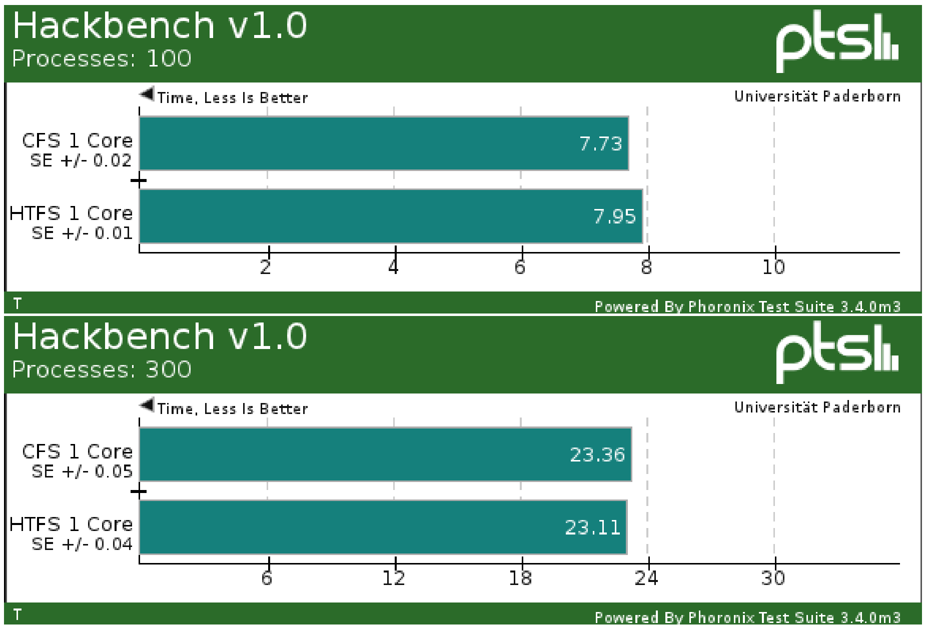


Figure 9: Hackbench results for 100 and 300 processes.

Processes	10	50	100	200	300
CFS	0.072	0.077	0.077	0.077	0.078
HTFS	0.071	0.078	0.080	0.079	0.077

Table 5: Hackbench: Seconds per process.

with more processes, while HTFS gets better. The HTFS task switches to the normal runqueues could influence these results because before this switch, the number of contextswitches should be really high. After the switch they are usually dropping to a level, which is desired by the task through leaving the CPU. The interaction of the kernel subsystems is very complex because of a lot of hardware and code aspects that can influence the speed of specific workloads. In the Hackbench test there are multiple subsystems involved, like the socket subsystem. HTFS might trigger some problems in these areas through different scheduling decisions that result in a performance loss. All together the performance loss of maximal 2.85% in the 100 processes test is acceptable.

Sysbench CPU test is a normal CPU intensive test. HTFS needs in average below 25 context switches per second. CFS has 600 to 1000 context switches per second. Figure 10 shows that HTFS performs slightly better than CFS. The reason should be the low number of context switches. But it is also visible that with such few context switches in general the difference in the result is very small.

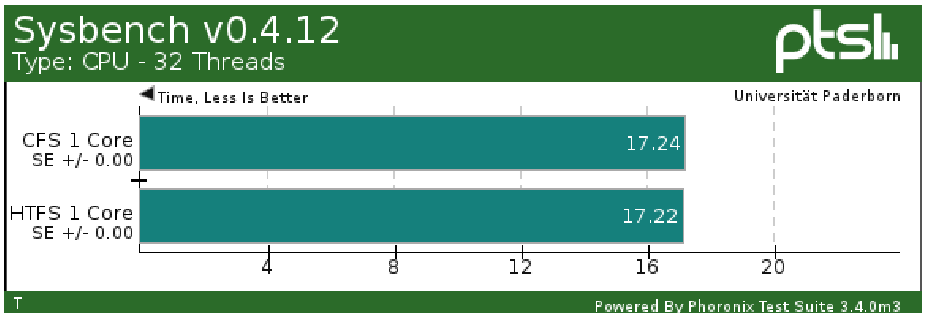


Figure 10: Sysbench CPU test results.

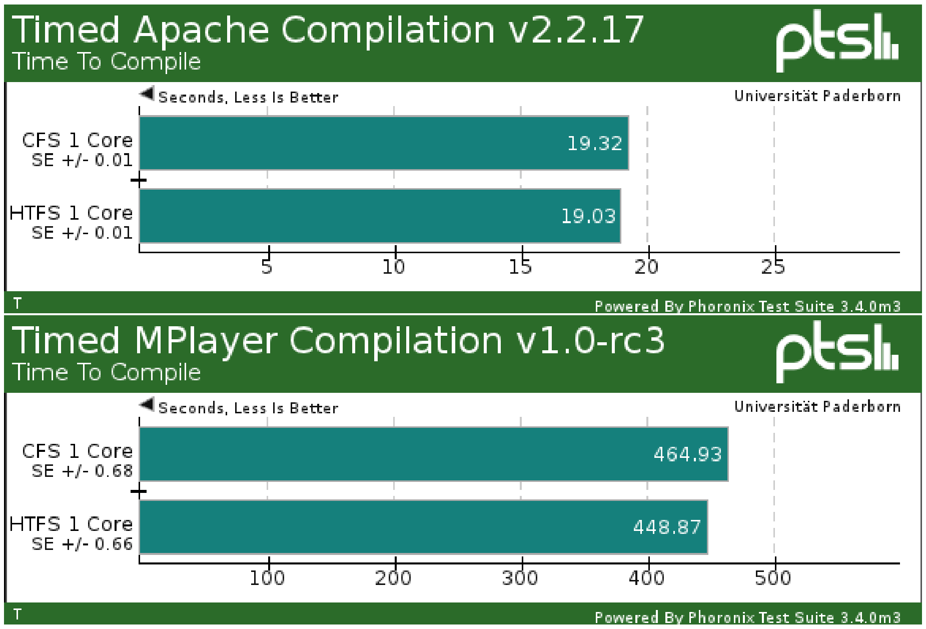


Figure 11: Apache and MPlayer compilation benchmark results.

Figure 11 shows the worst and the best results for the build benchmarks. The worst build was measured with the Apache Compilation. In this test HTFS is 1.5% faster than CFS. In the MPlayer build test HTFS reaches 3.45% higher performance than CFS, which is 26 seconds for a compilation time of 7.7 minutes. HTFS can perform extremely better in this type of workload because of the ability to increase the timeslice up to 1.07 seconds. This directly is visible in the number of contextswitches for both schedulers. CFS has an average of 332 switches while HTFS can reduce this to one third. Looking at the lowest number of contextswitches measured with the MPlayer build, CFS has 129 and HTFS has only 10, which is equivalent of each task having a timeslice of 0.1

seconds. CFS stays within latency for every task and serves strict fairness. Both restrictions make it impossible for CFS to lower the number of context switches, so it is difficult for CFS to compete with HTFS.

The other build test results are between 2.7% and 3.1% better than CFS. So this workload is a strength of HTFS.

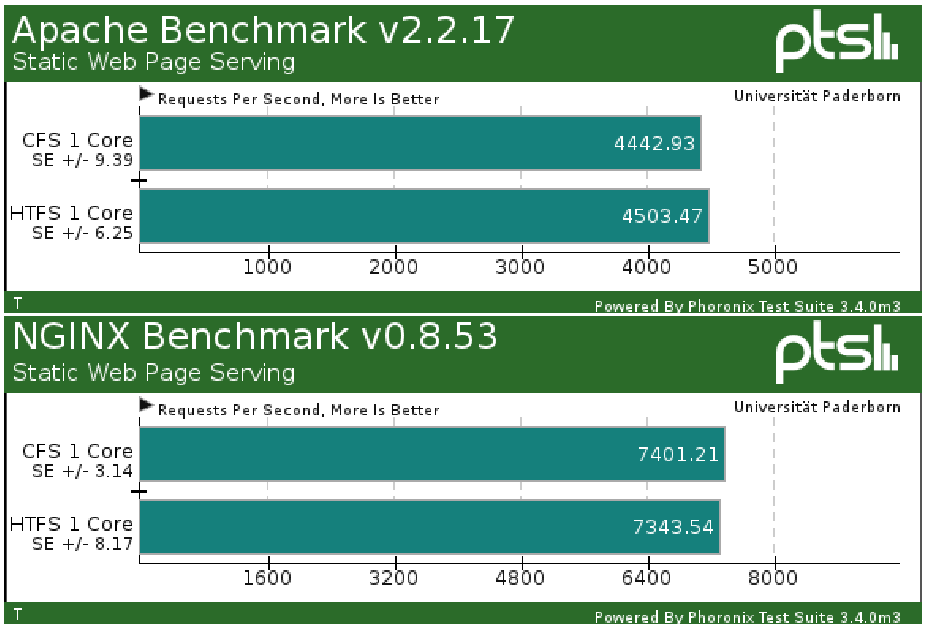


Figure 12: HTTP Server benchmark results.

HTTP server benchmarks are interesting because HTFS is better than CFS in the Apache Benchmark, while HTFS is worse with the NGINX webserver. The Apache Benchmark shows the usual image for HTFS and CFS. HTFS halfes the number of task switches of CFS, leading to 1.36% more requeuests per second.

Using NGINX as server results in 0.78% less requeuests per second. In Figure 13 you can see that HTFS produces much more contextswitches per second than CFS. NGINX makes use of the "sched_yield" system call, which triggers a schedule. We already saw the performance of both schedulers for this system call in Figure 3, where HTFS had more contextswitches. With this system call NGINX implements for example a spinlock and another function for the thread pool management. The locking with a spinlock makes the following scheduling decision important because there could be many threads waiting for a spinlock. All of them are still on the runqueue, so the scheduler can choose them all to run. CFS might have a higher chance to pick a non locked task than HTFS, which results in less contextswitches. Another possibility is that some code in NGINX profits from often continuation execution, even if the thread

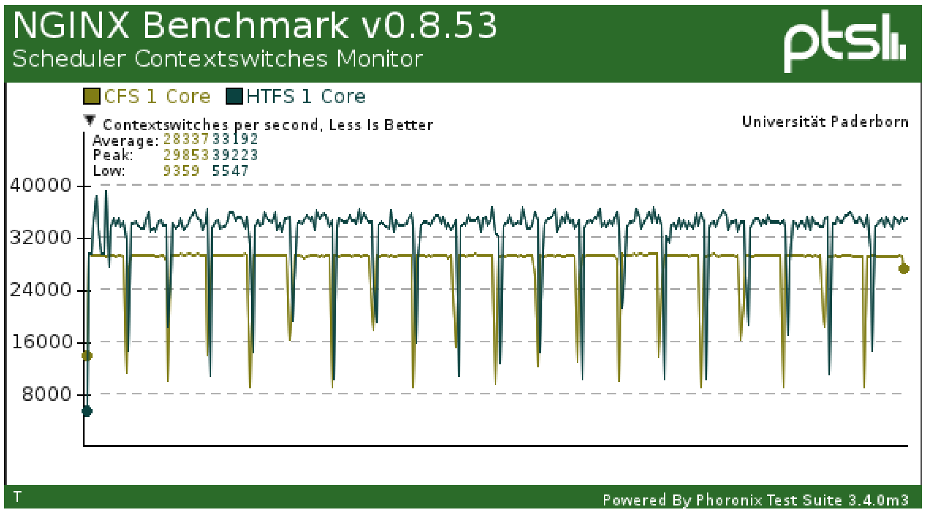


Figure 13: Context switches per second for NGINX Benchmark.

called "sched_yield". We saw the same behavior with the sysbench threads test although that was a synthetic benchmark.

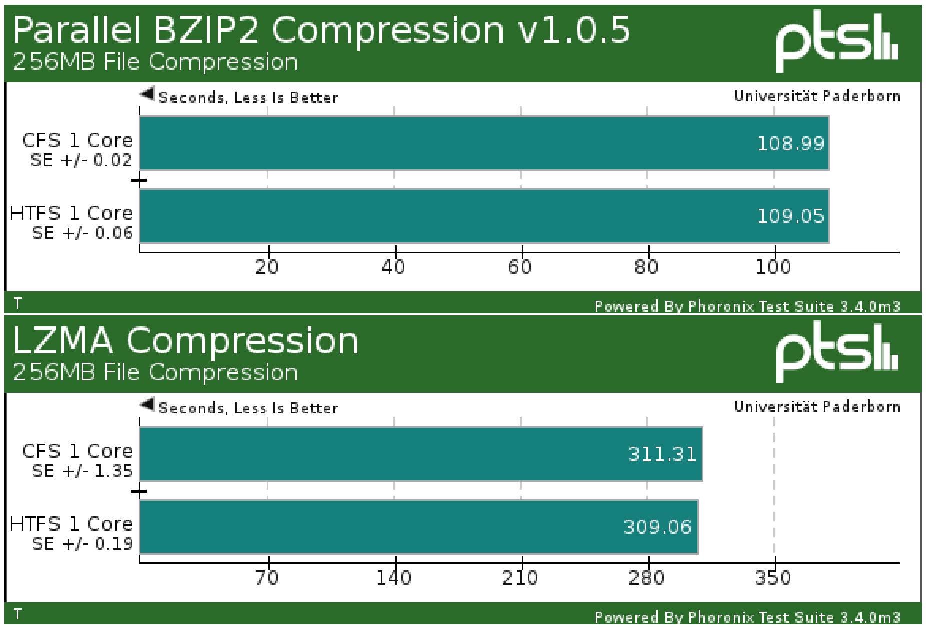


Figure 14: Parallel BZIP2 and LZMA Compression benchmark results.

Figure 14 again shows the two results, which are the worst or the best ones for compression benchmarks. HTFS is in the parallel BZIP2 benchmark 0.06% slower than CFS. In the LZMA test HTFS is 0.72% faster. Both result-differences are very small. The other compression benchmarks have even less difference. For the parallel BZIP2 the context switches are between 4 and 10 per second for both schedulers, so those tests are independent of the number of context switches. Thus CFS reaches low number of switches because all compression benchmarks are running on a single core system with only one active thread. This also explains why all compression results are similar for both schedulers.

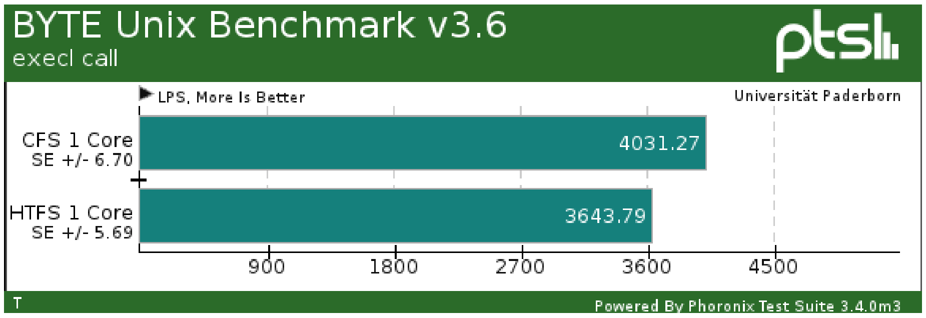


Figure 15: Unix-Benchmark “execl” result.

The result of the Unix-Benchmark’s “execl” benchmark is interesting because in this test HTFS is much worse than CFS although the average contextswitches are lower again. CFS has an average of 41, while HTFS only has 16 switches per second. But the results are contrary to the expectations, see Figure 15. HTFS makes 9.61% less loops per second than CFS in this test. It is difficult to find out why the “execl” system call because it is influenced by much kernel code that I don’t know of. The interaction of different kernel components is complex and difficult to analyze.

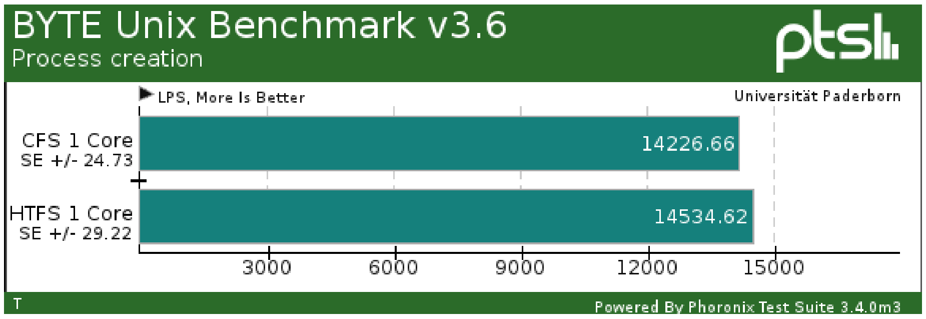


Figure 16: Unix-Benchmark Process Creation performance.

But HTFS is better in the other system calls tested, like Process Creation or Pipe throughput. Figure 16 shows that HTFS outperforms CFS in the speed of creating processes by 2.16%.

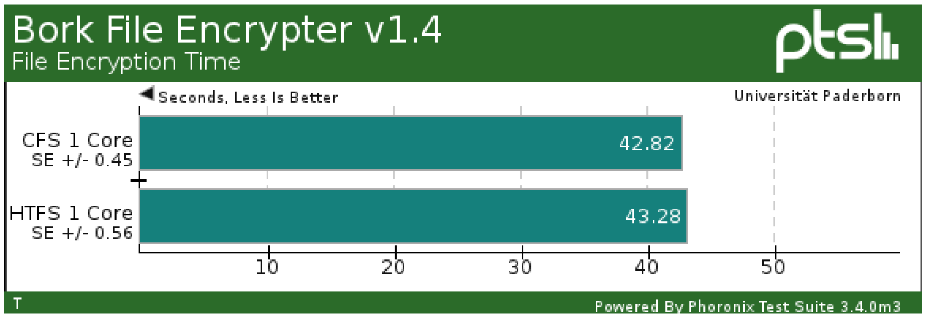


Figure 17: Bork File Encrypter result.

Figure 17 shows the Bork File Encrypter results. HTFS is slower by 1.07%. This test is very disk intensive. So it shows the performance of the scheduler for disk intensive workloads. Again, HTFS achieves less contextswitches per second, but it is still slower than CFS. So this could be a small problem with the kernel IO subsystem. For example the feature of HTFS for "Lock Acquire Retry" could delay some important IO processes. This is the only test using java which is a quite complex system. Java could have some workload characteristics that do not profit from the HTFS scheduling.

In most tests the number of context switches is reduced compared to CFS. As well we saw a higher scheduling code speed. So two of the goals are reached through HTFS. HTFS outperforms CFS in most of the benchmarks in a single core system, although there are some minor performance problems, which have to be further invastigated. You can find all test results in Section 5.4.

5.3 Multi Core Results

Testing on Multi Core systems adds another important factor, the load balancing. I could ignore this part for the single core tests, which showed the scheduler performance really good. However when switching to multi core systems I saw strange results in comparison to the single core tests. That is the reason why I measured all multi core tests with three different methods to calculate the load balancing metrics. These methods will be described in the next section. There are very interesting speedup results comparing the single core and dual core results. We are able to compare the single core and dual core results because they ran on the same platform. The Performance subsection will point out some interesting results about the different balancing methods and the influence of the hardware.

5.3.1 HTFS Balancing Methods

I tested with three algorithms to set the task load, runqueue load and cpu load.

std: This method tries to have the same balancing method like CFS. The average runqueue load is calculated nearly accurate by weighting the load over a timespan with the time this load was correct. The cpu load is the sum of all weights of the entities on the root runqueue. The remaining load that has to be moved is measured in the schedulable entitie's weight.

stdavg: This is nearly the same method as described above. Only the runqueue average is calculated by a very simple "moving average" method.

avg: This approach uses average loads for schedulable entities, which are calculated as simple moving average. Also the runqueue load average is calculated with a simple moving average. The CPU load average is equal to the root runqueue's load average. This approach interpretes the required load movement between CPUs as average load.

5.3.2 Fairness

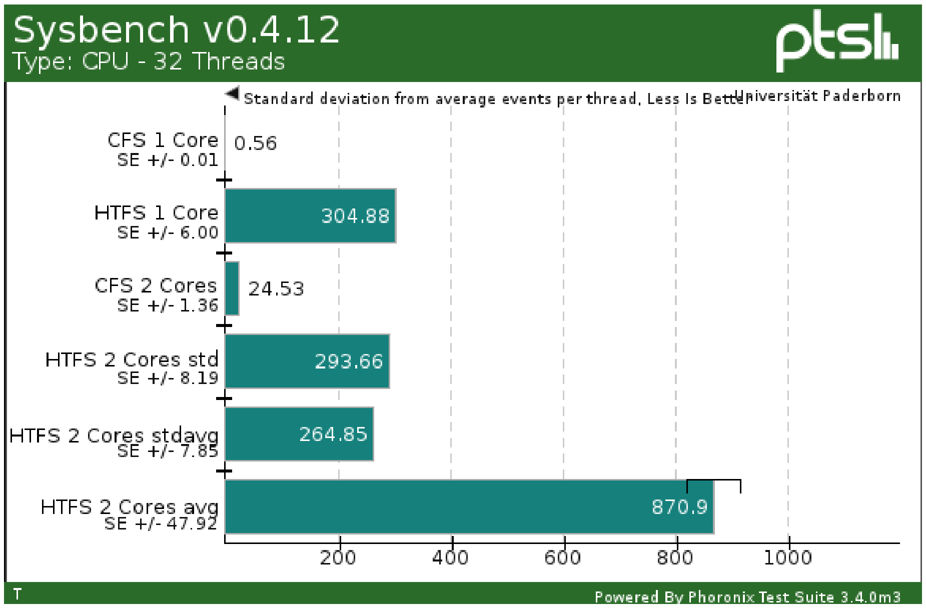


Figure 18: Sysbench standard deviation from average work done.

With load balance in action it also influences the fairness. You can see in Figure 18 the higher deviation from the average for all schedulers. CFS again has a very low deviation. The three balancing methods reach very different fairness values. The std and stdavg methods do reach a higher fairness than the single core result. The avg method fails to reach a balanced situation which results in a higher standard deviation, but still within a constant limit. This only reflects the fairness for this workload. Other workloads or mixed workloads might have different deviations that were not measured in any other test.

5.3.3 Speedup

The speedup of different applications is very dependent of the structure of these. But not only the application is an important factor, also the kernel with the scheduler has a big influence on the performance. In this section I only use the single and dual core results because both ran on the same platform. The comparison with the hexa core system would not give us any valid information because the CPU architectures are too different.

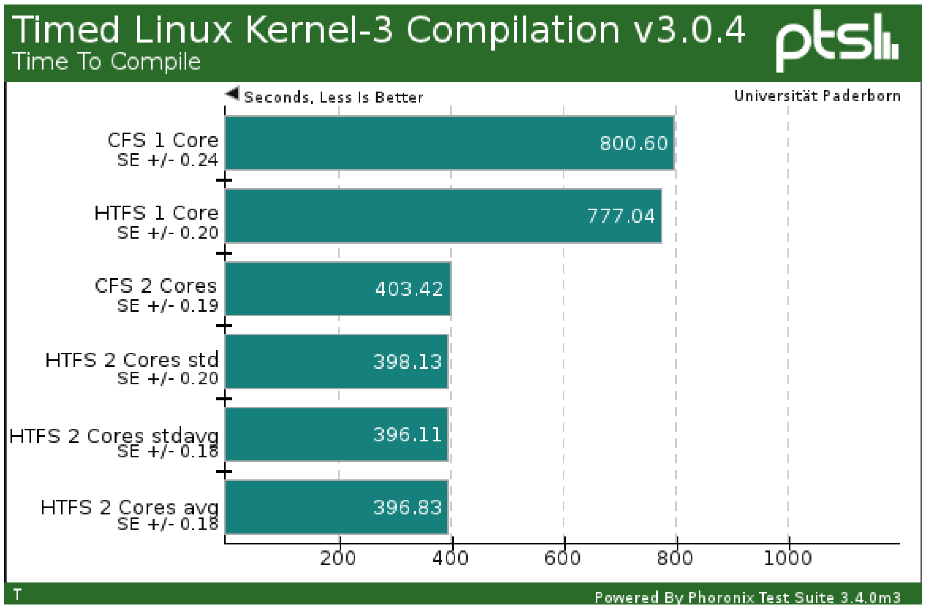


Figure 19: Linux kernel compilation results.

Figure 19 shows a nearly ideal scaling of a benchmark. The runtime is approximately halfed for the kernel compilation test. HTFS probably has bad balancing methods, which lead to non-optimal scaling. The number of context switches are slightly higher as in the single core tests. CFS has around 500 per second while all HTFS schedulers have around 300 per second. The number of load balancing calls is similar for all schedulers at 11 to 13 in average. Figure 20 shows the number of migrations per second. All HTFS schedulers have much more migrations than CFS has. That slows down all HTFS schedulers. With 203 task migrations per second, HTFS stdavg reaches the best result. Even with 94000 task migrations in average, the avg method is still faster than

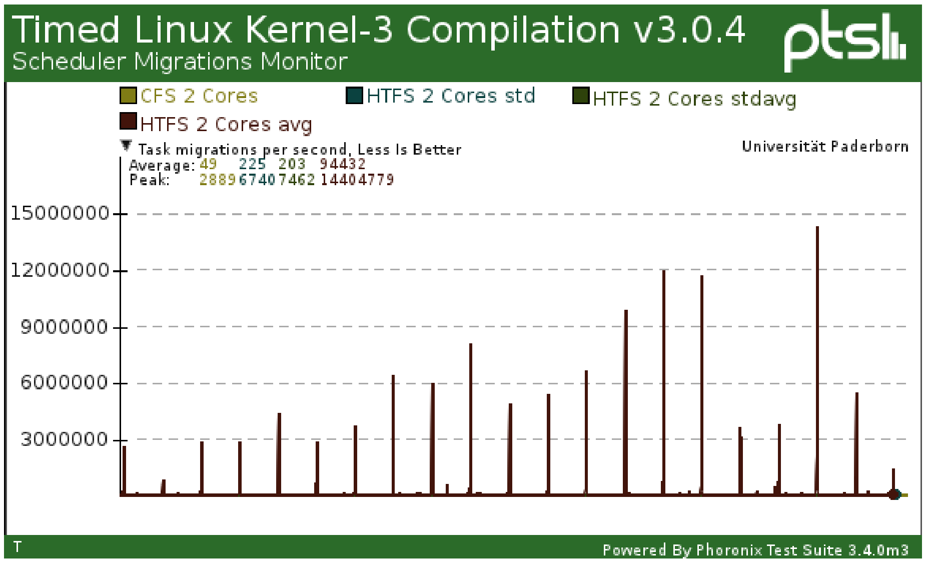


Figure 20: Task migrations monitor for Linux kernel compilation.

CFS and HTFS std. But this high number of migrations is an indication for a bad balancing. Still the avg method is better than the std method with its 225 migrations per second. This shows that the balancing decisions quality is important too, beside the number of task migrations.

But massive amounts of task migrations can improve the performance a lot. The Apache benchmark is an example where massive task migrations cause a much better performance. Figure 21 shows the results for the Apache benchmark and the number of task migrations. The first interesting point regarding those results is the bad scaling of CFS in the Apache benchmark from single core to dual core. All HTFS balancing methods reach better results. I picked CFS and the HTFS std balancing for an example to look at the task migrations per second and the influence on the requests per second in Figure 22. HTFS has nearly two times of the task migrations CFS has. The number of load balances are nearly the same. This indicates a problem with the calculation of the load that has to be moved to another CPU or the policy that long running tasks are prefered for task balancing. But it is also worth mentioning that even the original CFS balancing has problems here because the performance is really bad compared with all HTFS balancing operations and the single core performance, although the task migrations are extremely high.

To show the effort of the current scheduler into finding the best CPU for a new task, the process creation benchmark of the Unix-Benchmark is really interesting, Figure 23. Instead of a speedup this is actually a massive decrease in speed. The scheduler tries to find the best CPU to place a new task on. The scheduler decides on different factors which CPU to use. For example the

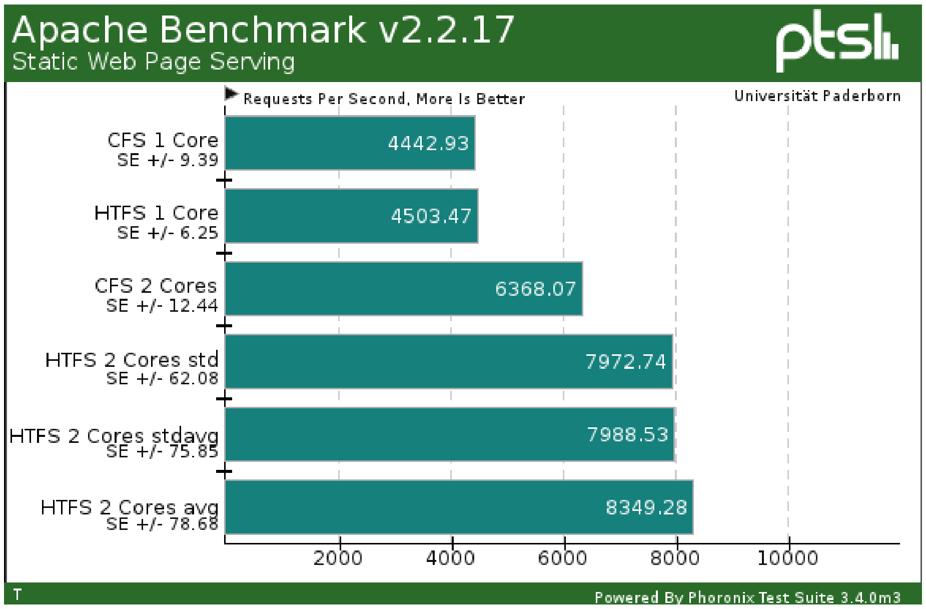


Figure 21: Apache Benchmark results.

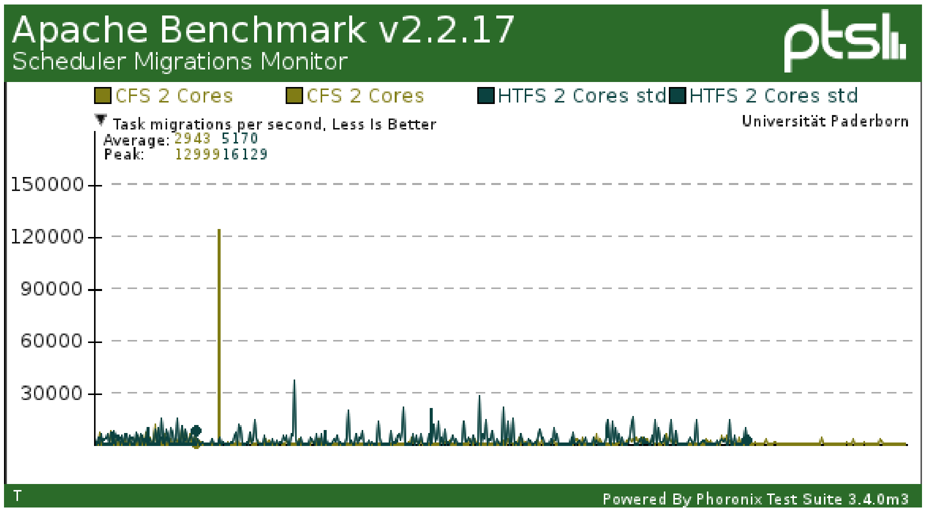


Figure 22: Apache Benchmark task migrations monitor.

power consumption is important and the current load of the CPUs. But also a possible pipe between the task and the child influences the decision, because grouping of tasks with a pipe between them is better because of CPU caches. This effort is understandable but it does cost much time, which results in a bad

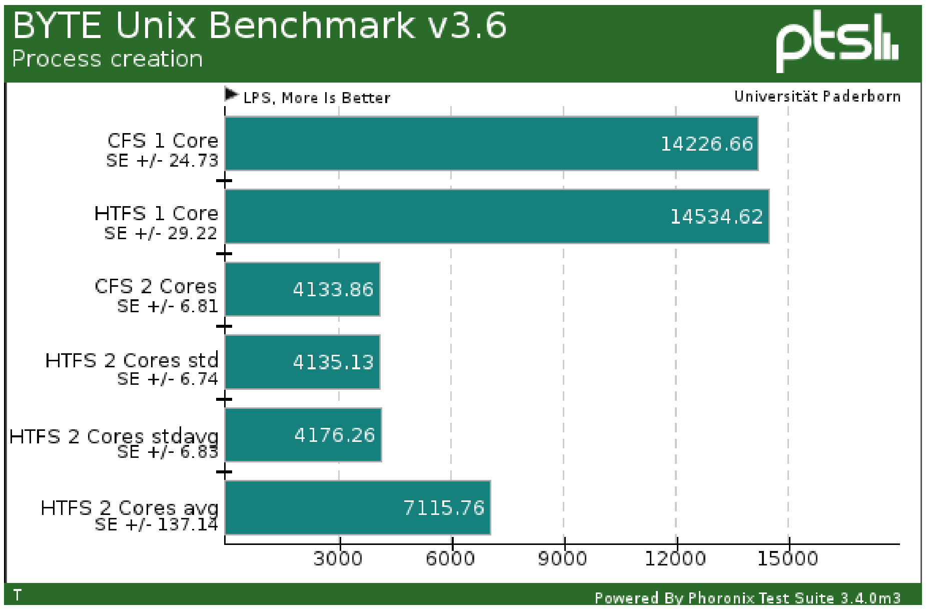


Figure 23: Unix Benchmark process creation results.

process creation performance for multi core systems. But especially on multi core server systems the number of created processes per second is increasing with every additional CPU. This results in a higher CPU usage through the system and less usable CPU time for other tasks. Interesting is again the much higher performance of the HTFS avg method through a massive number of task migrations.

5.3.4 Performance

In this section I describe some interesting influences of the balancing methods regarding the performance. Starting with the hard disk intensive Bork File Encrypter benchmark, you can see big performance differences between the dual core and hexa core systems. Figure 24 shows the results of Bork File Encrypter on multi core systems. On the dual core system all of the HTFS schedulers do not reach a good balance between CPUs, resulting in many task migrations and a worse encryption speed. On the hexa core system, all HTFS methods reach better results than CFS. Both systems reach better results compared to CFS than on the single core system. These results are really interesting because the benchmark is mainly limited by the performance of a single CPU, as the CPU usage shows. At the same time different balancing methods make noticeable performance differences.

Figure 25 shows the effect of bad balancing on the very good compilation performance of HTFS. The theoretical performance of HTFS on a dual core

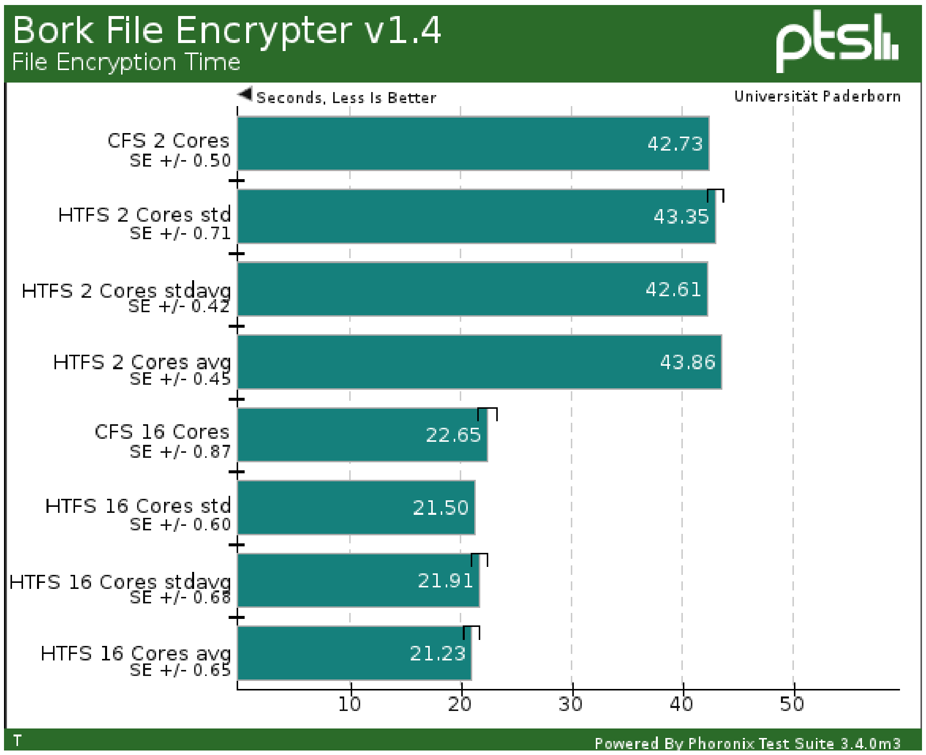
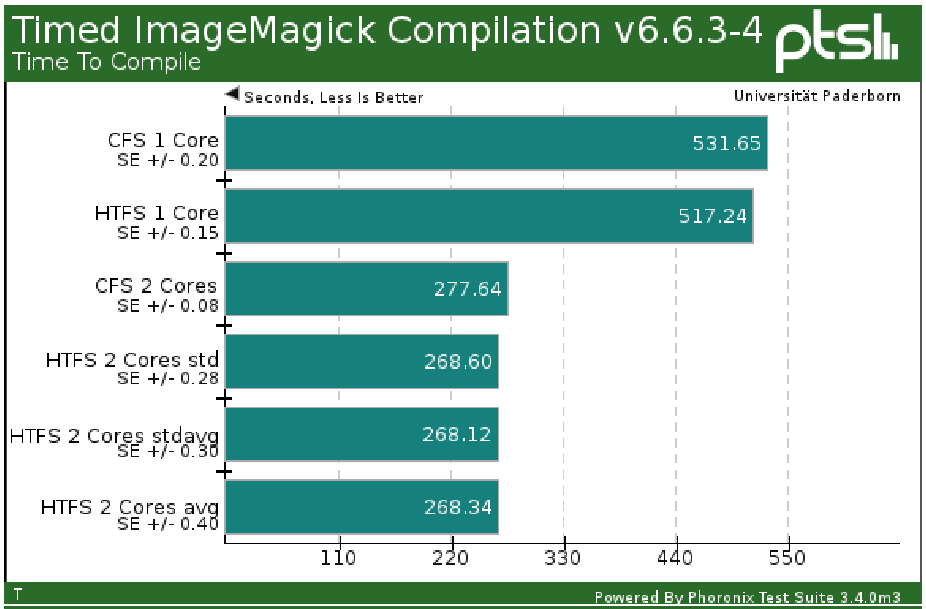


Figure 24: Bork File Encrypter multi core results.

system is much higher. The main aspect of HTFS is to replace the runqueue design, which is done per CPU. So with this design it is possible to reach the singlecore speedup for every CPU of the system. All multicore systems have hardware and software overhead, so on a dual core machine the scalefactor is not 2, but 1.91 with CFS. This sums up to a theoretical better dual core performance of HTFS of $1.91 \cdot 2.75\% = 5.25\%$. The 2.75% are the performance benefit of HTFS in the single core system. This leads to a possible result for ImageMagick Compilation of $263.06s$, which is reached by none of the HTFS balancing methods. Figure 25 shows that all HTFS schedulers produce much more task migrations per second than CFS does. This costs time and reflects the non optimal balancing situations.

The other compilation benchmarks have even more problems with the balancing. The performance benefit is partly less than in the single core tests, which is always directly visible in the number of task migrations or the CPU load.



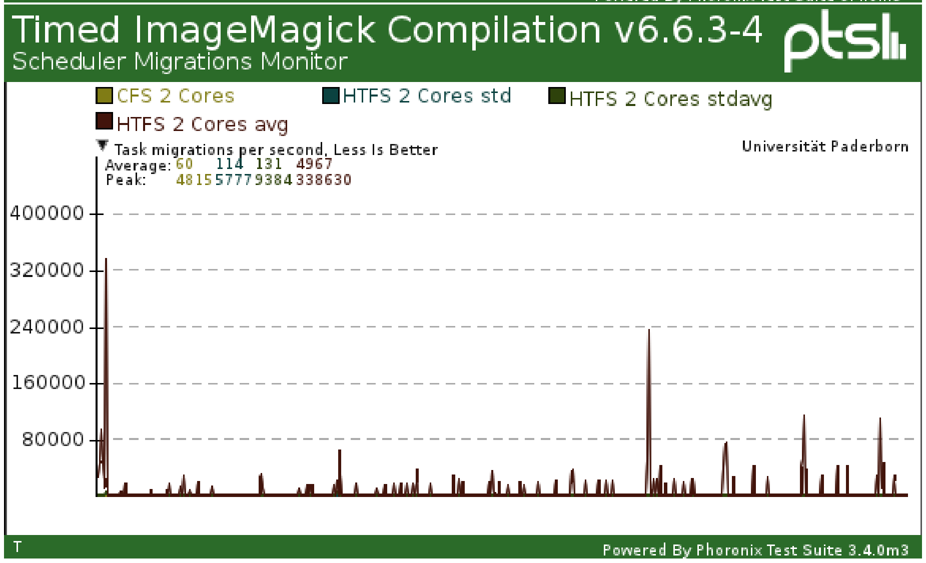


Figure 25: ImageMagick Compilation results.

Figure 26 shows such a case where a balancing method fails to reach equal load on all CPUs. The result is for the HTFS avg balancing method a CPU usage that does not reach 100%, so the compilation takes more time to finish. In the same test sometimes an active load balancing occurs. The active load balancing is triggered if the normal load balancing fails to reach an acceptable load difference. CFS did not need active balancings in any of the tests. This is

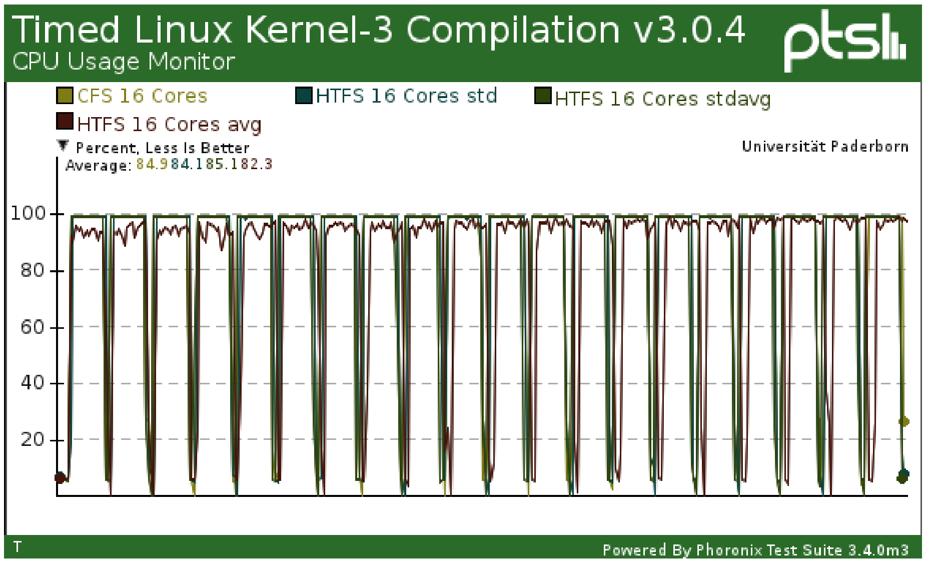


Figure 26: Kernel Compilation CPU usage.

another indicator for bad balancing methods of HTFS.

Figure 27 shows the 7-Zip multi core performance. This is a good example where the CFS balancing is worse than all of the HTFS balancing methods. You can see that the single core results are the same for both schedulers. On multi core systems HTFS gain the lead over CFS. This shows that in this workload CFS is not able to balance the load as good as HTFS does. CFS has an average of 14908 task migrations per second, while the number of context switches per second is nearly 7700. With only 3.7 created processes per second, the number of task migrations is much too high. HTFS can reduce the number of migrations with some of the balancing methods to nearly 10000, which is still very high, but sufficient for a noticeable performance increase. So there is much potential to increase multi core performance of CFS and HTFS.

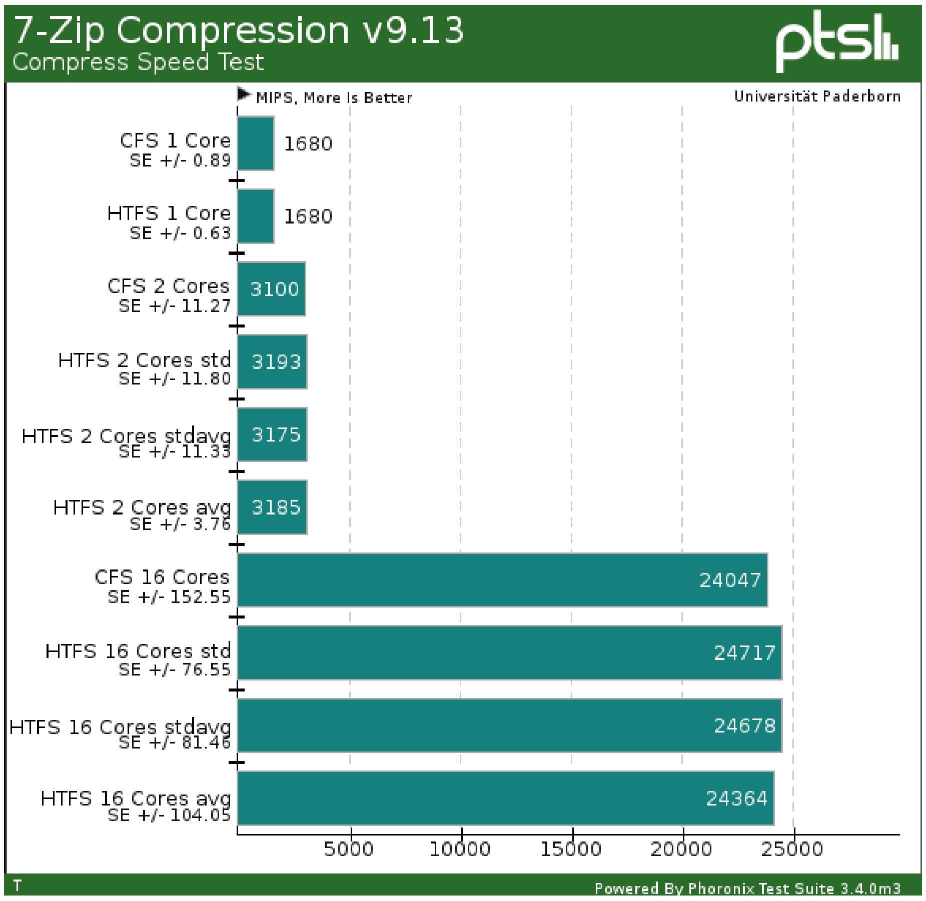
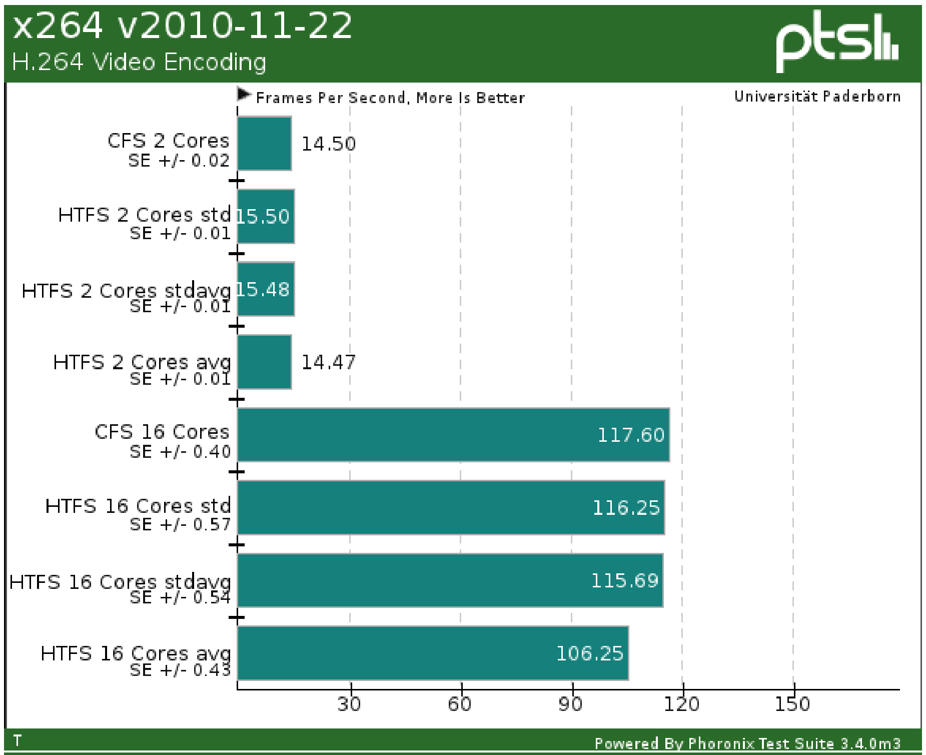


Figure 27: 7-Zip Compression results.

On the dual core system HTFS achieves partly significant higher performance. In Figure 28 you can see that the HTFS std and stdavg balancing can improve the CPU usage by 5% compared to CFS. The number of task migrations for those two methods is slightly higher than CFS. The avg method has an extremely high number of task migrations per second of 2 million. With this high number the CPU usage is equal to CFS and the frames per second are slightly below CFS's 14.5. On the hexa core system the number of task migrations for the std and stdavg methods is with 27000 and 34000 much more higher than the 23000 of CFS. Although the CPU usage of HTFS std and stdavg methods is again slightly higher than that of CFS, HTFS could not get better results than CFS. The avg method again fails completely through the extreme number of migrations. Through the two socket system some of the migrations are more expensive. Perhaps there are many task migrations between the two processors, which lead to a waste of time and worse results than CFS.



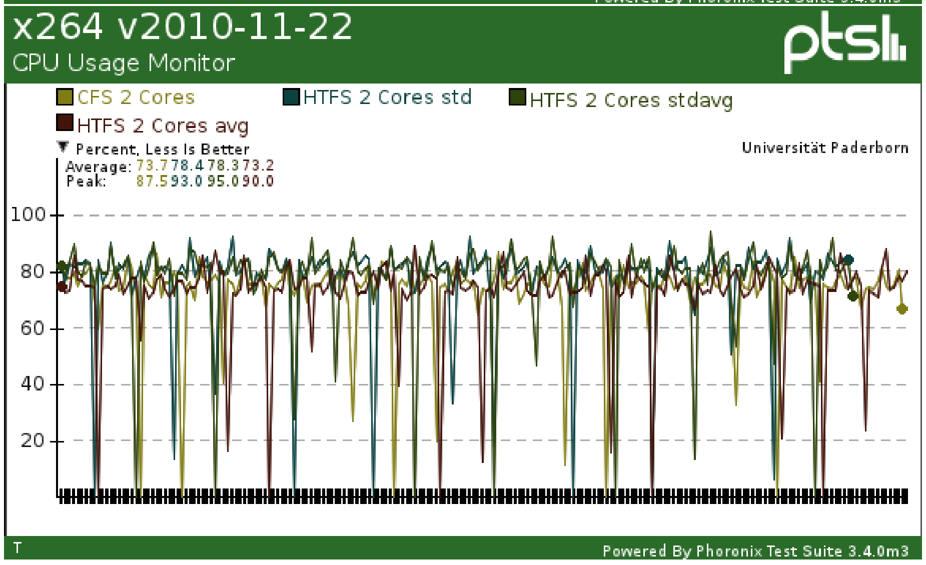


Figure 28: x264 results and CPU Usage monitor.

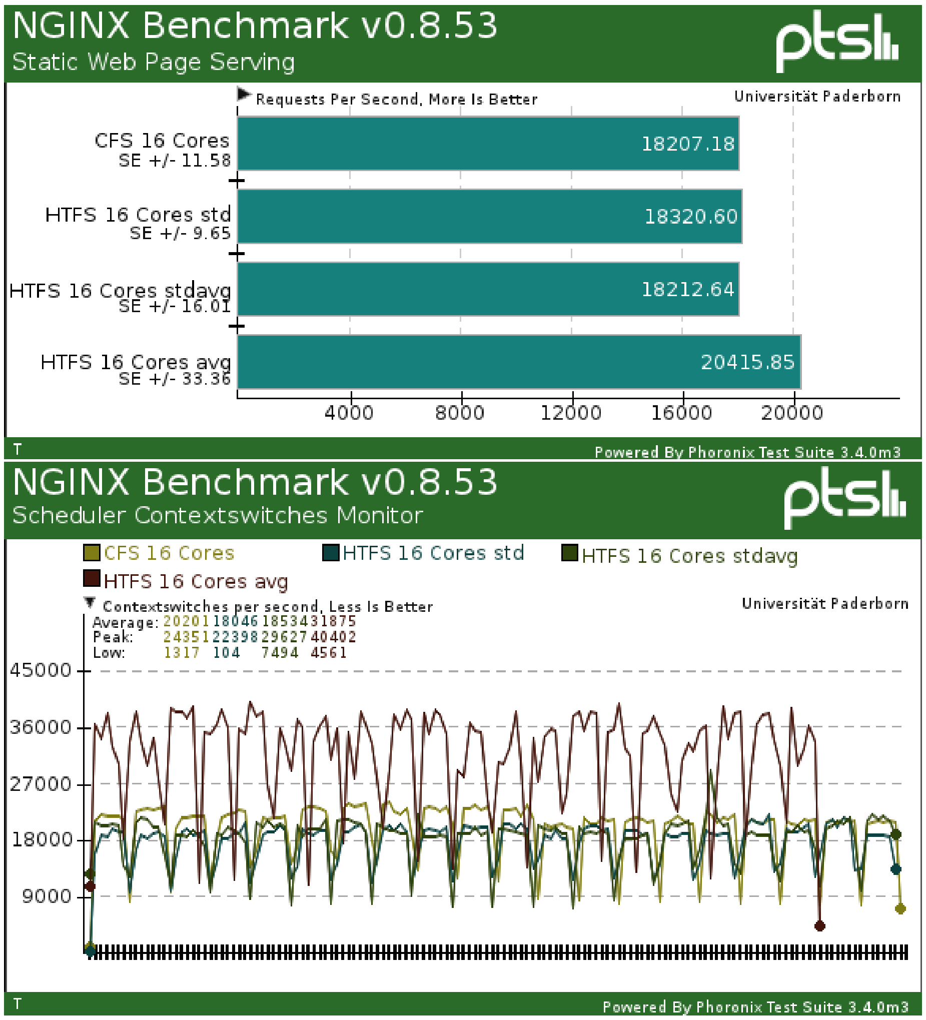


Figure 29: NGINX results and context switches monitor.

In some situations the massive number of task migrations can be positive. In Figure 29 you can see for the avg balancing method a much higher requeusts per second value. As already seen in the single core performance analysis, NGINX profits from a high number of context switches, perhaps due to the usage of the “sched_yield” system call. The number of context switches drop in this test for the avg method. This is a side effect of the immense number of task migrations, because some of them migrate running tasks, which triggers a contextswitch. All in all this leads to better results and higher CPU usage.

All HTFS balancing methods are still able to compete with CFS, although HTFS does not reach the possible performance in multicore systems. Most times the better performance of HTFS does not reach the theoretical performance. In some tests we also saw flaws with the current CFS balancing algorithm, where an extreme number of task migrations caused a huge performance increase. So there is much potential for CFS and HTFS to improve the performance on multi core systems.

5.4 Result Overview

I summarized all test results from all test platforms in three tables. The single core tests are presented in Figure 6, the dual core tests in Figure 7, and the hexa core tests are presented in Figure 8. I colored the results from best to worst. The percentage value shows the difference between HTFS and CFS results.

For a better general survey of how many tests CFS or HTFS were better, I created three tables. I excluded the sysbench standard deviation results from the tables because they do not show any performance. All Hackbench test runs together get one point because they are the same benchmark. Every other benchmark is worth one point. I only summarized those results where the scheduler was better than and not equal to the other scheduler. The hexa core system has 6 tests less than the dual and single core ones.

Single Core	**HTFS**
CFS better than ...	31.7%
... better than CFS	51.7%

Dual Core	**HTFS**		
	std	stdavg	avg
CFS better than ...	41.7%	20.0%	40.0%
... better than CFS	50.0%	66.7%	47.5%

Hexa Core	**HTFS**		
	std	stdavg	avg
CFS better than ...	36.7%	42.2%	44.4%
... better than CFS	63.3%	55.6%	50.0%

You can see that all variations of the HTFS scheduler are better in many tests, although the balancing methods did not work as expected. The stdavg method seems to be the best because it is better in much more tests than CFS. In the hexa core system the std method is the best one.

6 Conclusion

First we need to check the goals proposed for HTFS. All operations that are newly defined by HTFS are $\mathcal{O}(1)$ operations. I did not change the general structure of the load balancing methods. Still these functions are partly dependent of the number of tasks running because of some load balancing routines, which could iterate over all tasks on a runqueue. But HTFS alone reached this goal.

In the sysbench threads benchmark we could see the speed of HTFS compared with CFS. It is very fast even with high number of schedule calls. Also in other benchmarks HTFS was faster while having more contextswitches than CFS. So the goal that HTFS has to run with a low constant time for operations is also fullfilled.

The fairness of CFS is very strict. HTFS is by design not able to guarantee the same strict fairness. But the deviation is limited as seen in the benchmarks. Theoretical, HTFS also guarantees that all virtual runtimes are within constant limits because of the punishment for tasks, which leave the CPU before their timeslice is completely used. So the theoretical unfairness of HTFS is constant, which makes HTFS a fair scheduler with the own fairness model extension for task punishments.

Another goal was the reduction of contextswitches. We saw especially in the single core tests that the contextswitches were less than CFS's contextswitches. In many cases it lead to a better performance.

The last goal was the reduction of task migrations. HTFS definetly did not achieve this goal. We saw that some tests have less task migrations but most of them had much higher migrationrates. Beside the reduction of task migrations, the quality of those migrations is important too. A bad quality can lead to a CPU Usage lower than 100%, which was visible in some tests.

HTFS reaches a very good interactivity. In all of the tests the latency of HTFS was equal or better than CFS's latency. This shows the advantage of seperated handling of interactive tasks over the non explicit interactivity handling of CFS. But HTFS only uses a heuristic to differentiate between interactive and normal tasks. As soon as the heuristic makes in any situation a bad decision, it directly influences the interactivity. In none of the benchmarks such situation occured.

Although HTFS does not have a perfect balancing at the moment, all HTFS balancing methods are better in more tests than CFS. But especially the single core tests show the performance of HTFS, where it is most times better than CFS. The big strength of HTFS are compilation workloads, where it reaches up to 3.45% more performance. Also the 1.36% more requests per second delivered by HTFS is an interesting strength for apache webservers. Due to HTFS being only a replacement for the runqueue management, the performance differences could scale with a proper balancing algorithm. This could lead to over 10% higher compilation speed on modern servers.

Independent of the concrete HTFS implementation, also CFS has much potential regarding the balancing. We saw in multiple tests that CFS had trouble to reach a balanced situation over all CPUs. By improving the balancing algorithm, many servers could reach in some workloads about $1 - 5\%$ more performance. But improving a scheduler is always a difficult task because every code modification can change the overall performance immensly.

So at the end HTFS is an approach for scheduling that could be worth further development especially for servers. Also it should be possible to reach better multi core performance by implementing a proper balancing or by further investigation in the problems of the current balancing methods. With some keen tuning of the many constants in the code and performance optimization, the single core performance could be further increased about 0.1% or more. HTFS definitely is a design that is interesting for future schedulers.

7 Future Work

The major problem of HTFS is a bad balancing that should be corrected in future works. The solution of this problem has the potential to give HTFS a huge performance improvement in multi core systems.

Beside this there are some minor problems that need further work. For example the limits for timeslices should be set dependent of the systems performance and clock resolution. Also the punishment mechanisms should be set to the clock resolution. The taskgroup support is working currently but the performance and interactivity with taskgroups has to be measured and perhaps slightly corrected. Another minor open task is a more dynamic interactive timeslice calculation that depends on the number of currently running processes. In a scenario with 2 starting processes, the timeslices could be increased and the contextswitches decreased at the beginning.

So in future work HTFS will get a better balancing, some minor modifications to slightly adjust the behavior and some keen tuning and code optimizations to run at the highest possible speed.

A Results

Single Core			
Test	**CFS**	**HTFS**	
Hackbench 10	🟥 0.72	🟩 0.71	🟩 −1.39%
Hackbench 50	🟩 3.86	🟥 3.92	🟥 1.55%
Hackbench 100	🟩 7.73	🟥 7.95	🟥 2.85%
Hackbench 200	🟩 15.47	🟥 15.72	🟥 1.62%
Hackbench 300	🟥 23.36	🟩 23.11	🟩 −1.07%
Sysbench CPU	🟥 17.24	🟩 17.22	🟩 −0.12%
Sysbench Threads	🟩 74.31	🟥 74.53	🟥 0.30%
Sysbench Mutex	🟩 0.15	🟩 0.15	🟩 0.00%
Apache Benchmark	🟥 4442.93	🟩 4503.47	🟩 1.36%
Bork File Encrypter	🟩 42.82	🟥 43.28	🟥 1.07%
Apache Build	🟥 19.32	🟩 19.03	🟩 −1.50%
ImageMagick Build	🟥 531.65	🟩 517.24	🟩 −2.71%
Kernel Build	🟥 800.60	🟩 777.04	🟩 −2.94%
MPlayer Build	🟥 464.93	🟩 448.87	🟩 −3.45%
PHP Build	🟥 232.72	🟩 225.56	🟩 −3.08%
Unix-Bench Pipe	🟥 1592369.05	🟩 1600353.54	🟩 0.50%
Unix-Bench execl	🟩 4031.27	🟥 3643.79	🟥 −9.61%
Unix-Bench Process Creation	🟥 14226.66	🟩 14534.62	🟩 2.16%
CLOMP	🟩 0.98	🟩 0.98	🟩 0.00%
7-Zip	🟩 1680.00	🟩 1680.00	🟩 0.00%
Gzip	🟩 27.24	🟥 27.25	🟥 0.04%
LZMA	🟥 311.31	🟩 309.06	🟩 −0.72%
Parallel BZIP2	🟩 108.99	🟥 109.05	🟥 0.06%
FFmpeg	🟥 29.15	🟩 29.03	🟩 −0.41%
Fhourstones	🟩 6098.48	🟥 6085.08	🟥 −0.22%
x264	🟩 9.61	🟩 9.61	🟩 0.00%
C-Ray	🟥 748.54	🟩 747.79	🟩 −0.10%
NGINX	🟩 7401.21	🟥 7343.54	🟥 −0.78%
🟩 First best result			
🟥 Second best result			

Table 6: Table with all results for the single core tests. The percent value is the difference of the HTFS to CFS result in percent.

Dual Core							
Test	**CFS**	**HTFS std**		**HTFS stdavg**		**HTFS avg**	
Hackbench 10	0.47	0.48	2.13%	0.47	0.00%	0.46	−2.13%
Hackbench 50	2.41	2.61	8.30%	2.61	8.30%	2.39	−0.83%
Hackbench 100	4.79	4.97	3.76%	5.14	7.31%	4.94	3.13%
Hackbench 200	9.48	9.53	0.53%	9.53	0.53%	10.21	7.70%
Hackbench 300	14.22	14.23	0.07%	14.36	0.98%	15.44	8.58%
Sysbench CPU	8.58	8.57	−0.12%	8.57	−0.12%	8.58	0.00%
Sysbench Threads	47.16	26.14	−44.57%	25.51	−45.91%	23.60	−49.96%
Sysbench Mutex	0.73	0.70	−4.11%	0.72	−1.37%	0.47	−35.62%
Apache Benchmark	6368.07	7972.74	25.20%	7988.53	25.45%	8349.28	31.11%
Bork File Encrypter	42.73	43.35	1.45%	42.61	−0.28%	43.86	2.64%
Apache Build	11.71	11.74	0.26%	11.62	−0.77%	11.81	0.85%
ImageMagick Build	277.64	268.60	−3.26%	268.12	−3.43%	268.34	−3.35%
Kernel Build	403.42	398.13	−1.31%	396.11	−1.81%	396.83	−1.63%
MPlayer Build	234.51	230.64	−1.65%	230.34	−1.78%	230.80	−1.58%
PHP Build	126.65	124.61	−1.61%	123.93	−2.15%	122.97	−2.91%
Unix-Bench Pipe	1545731.14	1533719.65	−0.78%	1547514.46	0.12%	1537571.60	−0.53%
Unix-Bench execl	2123.85	2026.53	−4.58%	2116.67	−0.34%	3667.24	72.67%
Unix-Bench Process Creation	4133.86	4135.13	0.03%	4176.26	1.03%	7115.76	72.13%
CLOMP	1.81	1.81	0.00%	1.81	0.00%	1.81	0.00%
7-Zip	3100.00	3193.00	3.00%	3175.00	2.42%	3185.00	2.74%
Gzip	26.32	26.37	0.19%	26.25	−0.27%	26.35	0.11%
Parallel BZIP2	55.85	55.91	0.11%	55.90	0.09%	55.87	0.04%
FFmpeg	27.60	27.62	0.07%	27.60	0.00%	27.88	1.01%
Fhourstones	6077.46	6090.32	0.21%	6100.61	0.38%	6098.97	0.35%
x264	14.50	15.50	6.90%	15.48	6.76%	14.47	−0.21%
C-Ray	372.41	372.73	0.09%	372.65	0.06%	374.32	0.51%
NGINX	14443.98	14295.79	−1.03%	14325.53	−0.82%	14384.05	−0.41%
First best result							
Second best result							
Third best result							
Fourth best result							

Table 7: Table with all results for the dual core tests. The percent value is the difference of the HTFS to CFS result in percent.

Hexa Core (Hyperthreading)							
Test	**CFS**	**HTFS std**		**HTFS stdavg**		**HTFS avg**	
Hackbench 10	0.09	0.09	0.00%	0.09	0.00%	0.09	0.00%
Hackbench 50	0.41	0.39	−4.88%	0.41	0.00%	0.37	−9.76%
Hackbench 100	0.79	0.81	2.53%	0.83	5.06%	0.73	−7.59%
Hackbench 200	1.57	1.61	2.55%	1.66	5.73%	1.49	−5.10%
Hackbench 300	2.37	2.40	1.27%	2.47	4.22%	2.32	−2.11%
Apache Benchmark	16511.69	17835.38	8.02%	17539.62	6.23%	18218.89	10.34%
Bork File Encrypter	22.65	21.50	−5.08%	21.91	−3.27%	21.23	−6.27%
Apache Build	21.26	21.28	0.09%	21.34	0.38%	22.52	5.93%
Kernel Build	64.08	63.49	−0.92%	63.52	−0.87%	64.43	0.55%
MPlayer Build	27.20	26.76	−1.62%	26.70	−1.84%	26.07	−4.15%
PHP Build	24.47	24.41	−0.25%	24.43	−0.16%	25.44	3.96%
Unix-Bench Pipe	1523578.06	1513376.63	−0.67%	1512431.66	−0.73%	1494763.51	−1.89%
Unix-Bench Process Creation	7951.13	8001.91	0.64%	8005.41	0.68%	10251.66	28.93%
CLOMP	0.22	0.21	−4.55%	0.21	−4.55%	0.22	0.00%
7-Zip	24047.00	24717.00	2.79%	24678.00	2.62%	24364.00	1.32%
Gzip	20.82	20.85	0.14%	20.84	0.10%	21.06	1.15%
LZMA	185.27	184.34	−0.50%	184.16	−0.60%	184.32	−0.51%
Parallel BZIP2	6.75	6.74	−0.15%	6.76	0.15%	8.09	19.85%
Fhourstones	8014.73	8033.65	0.24%	8037.17	0.28%	8016.96	0.03%
x264	117.60	116.25	−1.15%	115.69	−1.62%	106.25	−9.65%
C-Ray	43.38	43.65	0.62%	43.72	0.78%	44.58	2.77%
NGINX	18207.18	18320.60	0.62%	18212.64	0.03%	20415.85	12.13%

First best result
Second best result
Third best result
Fourth best result

Table 8: Table with all results for the 16 core tests. The percent value is the difference of the HTFS to CFS result in percent. There are some tests missing because of incompatible software.

B Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe sowie ohne Benutzung anderer als der angegebenen Quellen angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Paderborn, 1. November 2011

Unterschrift

References

[1] "Top 500 supercomputers operating system family share for 06/2011." Website, 2011. http://top500.org/stats/list/37/osfam.

[2] "Operating system share by groups for sites in all locations january 2009." Website, 2009. https://ssl.netcraft.com/ssl-sample-report//CMatch/osdv_all.

[3] W. Maurer, *Professional Linux Kernel Architecture.* John Wiley & Sons, 2008.

[4] "Linux 2.6.38 documentation sched-design-cfs.txt." Linux documentation, 2011. Documentation/scheduler/sched-design-CFS.txt.

[5] M. Cesati and D. P. Bovet, *Understanding the Linux Kernel.* O'Reilly Media, third ed., 2006.

[6] "Linux 2.6.38 documentation sched-domains.txt." Linux documentation, 2011. Documentation/scheduler/sched-domains.txt.

[7] "Linux 2.6.38 documentation sched-nice-design.txt." Linux documentation, 2011. Documentation/scheduler/sched-nice-design.txt.

[8] M. Galbraith, "Automated per tty task groups." Mailinglist, 2010. http://marc.info/?l=linux-kernel&m=128978361700898&w=2.

[9] I. Molnar, "Completely fair scheduler." Website, 2007. http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt.

[10] C. Kolivas, "Brain fuck scheduler." Website, 2009. http://ck.kolivas.org/patches/bfs/bfs-faq.txt.

[11] I. Molnar, "O(1) scheduler." Website, 2002. http://people.redhat.com/mingo/O(1)-scheduler/README.

[12] J. Nieh, C. Vaill, and H. Zhong, "Virtual-time round-robin: An o(1) proportional share scheduler," Department of Computer Science Columbia University, USENIX, 2001.

[13] H. Franke, S. Nagar, M. Kravetz, and R. Ravindran, "Pmqs: Scalable linux scheduling for high end servers," IBM Thomas J. Watson Reasearch Center, USENIX, 2001.

[14] M. Kravetz, H. Franke, S. Nagar, and R. Ravidran, "Enhancing linux scheduler scalability," in *Proceedings of the Ottawa Linux Symposium*, IBM Thomas J. Watson Reasearch Center, USENIX, 2001.

[15] B. Saha, A.-R. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. L. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, A. Rohillah, D. Carmean, and J. Fang, "Enabling scalability and performance in a large scale cmp environment," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, (New York, NY, USA), pp. 73–86, ACM, 2007.

[16] M. Larabel, "Phoronix test suite." Website, 2011. http://www.phoronix-test-suite.com.