Study Thesis

Vector-Based Scheduling for the Core2-Architecture

Eric Hoh

Advisor:
Prof. Dr. Frank Bellosa
Dipl.-Inform. Andreas Merkel

University of Karlsruhe, Germany

January 20, 2009

Abstract

Contemporary operating systems use schedulers which select the next task to run due to its priority and/or its workload. In the past it was hard or even impossible to get more information about the tasks, but most modern architectures have performance monitoring counters (pmc) with which you can get much better information about task's characteristics in terms of usage of a CPU's functional units and the machine's bus-system. The scheduler can use this information to schedule similar tasks so that they keep a distance in time as well as in space which means that two similar tasks do not run at the same time on different cores (distance in time) or consecutively on the same core (distance in space). The purpose of this is the reduction of hotspots and the avoidance of stall cycles due to busy shared CPU units.

In this thesis we present an implementation which adds this functionality to the current Linux kernel scheduler. Additionally, the scheduler's migration-mechanism is modified to consider the task's characteristics while deciding which task to migrate to maximize the dissimilarity between the tasks in each run queue. We also implement a CPU frequency and voltage scaling system based on the information about the tasks' characteristics we get from the performance monitoring counters.

Contents

Chapter 1

Introduction

1.1 Scheduler

Today's operating systems still use schedulers which choose the next task to run because of its priority and/or its workload. These simple methods were extended to consider characteristics like interactivity measured by the difference between the time a task could have been used due to its time slice and the time it has really used. This method has been used by the old Linux scheduler and gives you only a very vague idea of the task's characteristics. For a more detailed profile you could use another approach, for example a syscall a task could use to tell the operating system what requirements it has, but therefore all applications must be modified to use this syscall. Furthermore programs could misuse this feature. If you know how the scheduler works and how it distributes the system resources among the running tasks you can fake your profile to get more system resources.

1.2 Performance Counter and Activity Vector

In this thesis we want to use performance monitoring counters (PMC), which are integrated in most modern CPU architectures. These counters deliver information about how often certain events such as the number of instructions, the number of branches or the number of cycles while the bus-queue is empty, occur. These numbers are put together to a so-called task activity vector or just activity vector (AVEC), which describes the characteristics of a task, a run queue or any other type of job or set of jobs. These AVECs are not only preciser than the metrics the scheduler has used yet, but they also do not depend on any support from userspace applications as syscalls would do.

1.3 Goals

1.3.1 Reducing Hotspots

With the help of these AVECs the scheduler can make better decisions. It can take care that whenever two following tasks are too similar another, more dissimilar, task is scheduled between them so that different CPU units (integer/floating-point arithmetic, branch prediction, caches, ...) are stressed during a certain time interval. This leads

to a better distribution of power consumption on the CPU die and therefore reduces hotspots on the die which always appear when the same units are constantly stressed. This could lead to unstable behaviour or even to a complete failure of the CPU although its overall temperature and therefore the temperature which is measured by the "ondie" temperature sensors is not critical.

1.3.2 Increasing Performance

Another benefit of this kind of scheduling has become relevant with the introduction of multicore CPUs. Those CPUs have almost independent cores but there are often some units left which must be shared between the cores. For Core2-Architecture these shared units are the L2 cache[1] and the frontside bus. That means all instructions which does not use memory or other hardware attached to the frontside-bus can be executed completely independently but whenever more than one running task needs a shared unit all but one of them have to wait. Core2's L2 cache is dual-ported, which means that both cores could use it concurrently, but you should remember that the L2 size is very limited in comparison to the RAM. Therefore it could result in a small performance benefit if you do not schedule two cache intensive tasks at the same time.

The knowledge of the task's requirements can help the scheduler to choose the tasks in a way that preferably only one running task needs the L2 cache, the memory or the the bus. This can reduce the number of memory accesses caused by a full L2 cache and stall cycles provoked by occupied shared CPU units.

1.3.3 AVEC-Based Frequency- and Voltage-Scaling

Today there are three commonly used classes of algorithms for frequency and voltage scaling in the Linux kernel. These algorithms are encapsulated in so-called governors, which are just sets of functions with a defined interface. The simplest class (performance and powersafe governor) sets the frequency and voltage statically to the highest/lowest possible value. The second class (ondemand governor) sets the values dynamically according to the CPU's average load in a defined period of time. The problem is, that this algorithm needs some time to adjust to new loads. If you let two tasks run, one which causes high load and one which does not, it can happen that the algorithm reduces the CPU frequency when the high-load task is scheduled and increases the speed when the low-load task is scheduled because of the delay with which the load is measured. The third class does not make any decisions on its own but provides an userspace-interface.

Using AVECs instead for choosing the CPU speed makes the algorithm task-sensitive so that the CPU is switched to highest speed just before the high-load task is scheduled and the other way round. A problem with this kind of scaling could be the delay for changing the CPU voltage if the tasks' timeslices get too short, but normally choosing a very short timeslice is not a good idea for most cases anyway.

1.4 Implementation

We implement AVECs for the Core2 Architecture and modify the scheduler to use this additional information. The modified scheduler considers the previously running task

[1] Actually the Core2Quad consists of two Core2Duo dies "glued together". They have two separate L2 caches shared only among the cores on the same die.

on the same core and the currently running tasks on the other cores while deciding which task to run next. Additionally we adapted the migration mechanism for the use of AVECs. At last we implement a CPU frequency governor which uses AVECs to choose the frequency/voltage.

Chapter 2

Background

2.1 Completely Fair Scheduler

Since version 2.6.23 the Linux kernel scheduler and its infrastructure has been changed almost completely. With the new infrastructure the policies such as CFS or the RT-Policy has been encapsulated into so called scheduling classes. This does not make the policies modular in terms of kernel modules. They cannot be loaded at runtime, but it has become much easier to integrate new policies and use them together with the existing ones. The idea behind CFS is very simple. Each task should get the same amount of CPU time. The task which should run next is determined by the difference between the time a task has run and the minimum runtime of a task in the runqueue. (this difference is called "entity_key") The CFS uses a rbtree, sorted by this entity_key, instead of a classical runqueue.

Additionally, scheduling groups have been integrated into the CFS. You can group tasks by the user who has started them or you can group them manually. These groups can consist of tasks or other groups. The scheduler goes through the tree and chooses the leftmost and therefore fairest-to-schedule entity.[1] Preserving this group mechanism makes it necessary to modify the CFS class on the lowest level, because the code contains a caching mechanism which is updated whenever the tree is searched. After searching for the fairest task once, the caching mechanism would lead you to the next group instead of to the next entity in the same group. [1, 9].

2.2 Task Activity Vectors

Our main source of inspiration has been Andreas Merkel's and Frank Belossa's paper about Task Activity Vectors [13]. We use the same approach of Activity Vectors to characterize a task's usage of functional units (FU). The most important changes have been provoked by the change of infrastructure. The switch from the old O(1) Linux scheduler to the new Completely Fair Scheduler makes changes necessary because of its completely different data structures.

Moreover, in contrast to the old scheduler the CFS has no explicit protection against starvation, because it does not need it. CFS's runqueue is sorted by the entity_key which gets smaller while a task is not scheduled[2]. The algorithm always chooses the

[1] a scheduling entity (SE) could be either a task or a group

[2] Actually the key does not get smaller, but the other tasks' keys get bigger when scheduled.

task with the smallest key so that it is just a matter of time until the task is scheduled. But with the implementation of runqueue-sorting it becomes important again, because these algorithms ruin CFS' implicit protection completely.

The second change is about the CPU architecture. They have used a Pentium 4 CPU for their paper. For our paper we use a Core2 CPU, which gives us at least an additional core, a shared L2 Cache and a much smaller number of performance monitoring counters. Hyperthreading is also not supported by the new architecture. Because of that we could simplify the AVEC part. The increased number of cores and the shared cache must also be considered by the scheduler while making its decision, because of them some instructions cannot be executed concurrently, although we have multiple cores.

For the Core2 architecture we cannot find any plan which describes the physical layout of the CPU die. But these plans would be necessary to run an accurate temperature simulation. The simulation would show us how much and where on the die heat is produced. The place is important because heat does not stay where it has been produced but spread throughout all neighbouring units. We have decided not to run these simulations because the results would not be worth the effort. Besides, the hotspot behaviour has been evaluated for the Pentium 4 in the above mentioned paper so that we can assume that it also works on the Core2 architecture considering that the missing information about the die layout lets the algorithms work less efficient.

We also add AVEC-based frequency and voltage scaling, which should react faster and more specific to thread switches in terms of choosing the frequency and voltage by the task's characteristics and not by the global load. This load would be most likely the load of the previous task and not the current task's one.

Chapter 3

Proposed Solution

3.1 Implementation Environment

We use the Linux Kernel 2.6.26 "Vanilla" in the 64bit version on a Core2Duo Dual-Core CPU for our work. We choose the 64bit version because all Core2 CPUs supports this and it makes some workarounds for 64bit calculations in the kernel unnecessary and therefore should improve the speed of our implementation.

We put most AVEC-stuff into an own module but that was not possible for some parts which must be integrated directly into the scheduler's migration- and task-picking-algorithms. These parts were implemented in a way that the scheduler uses the unmodified algorithms as long as the AVEC-module is not loaded and switches to the AVEC-code thereafter.

The frequency/voltage scaling code is encapsulated in a governor for the kernel's cpufreq-module. This makes the implementation easy but it is also the reason why we cannot change the frequency during scheduling. The cpufreq-module has not been designed for that purpose and so it does not work if called from there. We integrate a workaround which uses a kernel-thread to switch the frequency, but this method suffers from a very bad performance because the kernel-thread must be executed between every two tasks. A better solution would be setting the CPU's registers directly.

3.2 Activity Vectors

For our purpose the activity vector should give us information about which of the CPU's functional units are used by a task. Therefore we have to determine which instructions the task uses. For determining the cache and bus usage we do not count instructions, because there is no special instruction to use the cache and bus transactions must not be triggered by explicit instructions (e.g. prefetcher). Instead we count the cycles in which they are busy. Some values can be counted directly, some others must be calculated.

All Instructions This value is used only as reference for the other instruction events.

$$instr_{all} = \#event_{instr\_retired\_any}$$

Floating Point, MMX, SSE, Branch Instructions The CPU has special events to count

these instructions directly.

$$instr_{x87} = \#event_{x87\_ops\_retired\_any}$$

$$instr_{mmx} = \#event_{simd\_instr\_retired}$$

$$instr_{sse} = \#event_{simd\_inst\_retired\_any}$$

$$instr_{branch} = \#event_{br\_inst\_retired\_any}$$

Non-Memory Instructions We count these instructions only to calculate the number of integer instructions. This value is neither used for scheduling, nor for frequency scaling.

$$instr_{nomem} = \#event_{instr\_retired\_other}$$

Integer Instructions This value is often incorrect or at least imprecise due to the number of needed values and some overlapping of memory and MMX/SSE instructions. (3.6)

$$instr_{int} = instr_{nomem} - instr_{x87} - instr_{mmx} - instr_{sse} - instr_{branch}$$

All Cycles This value is only used as reference for the following events. It is not used for scheduling or frequency scaling.

$$cycles_{all} = \#event_{cpu\_unhalted\_clk}$$

L2 Cache Cycles We count the number of cycles in which the L2 cache is busy, because there are no special instructions to access the cache.

$$cycles_{L2} = cycles_{all} - \#event_{l2\_no\_req}$$

Bus Cycles We count the number of cycles in which the bus is in use, because not every bus usage is triggered by an explicit instruction.

$$cycles_{Bus} = cycles_{all} - \#event_{busq\_empty}$$

3.3 Special Data Structures and Parameters

We implemented some special datastructures. Some of them have been implemented only to test the algorithms, others are used for scheduling or migration.

3.3.1 avec_boosted data structure

This value tells us how fair a task has been treated in terms of being scheduled when it has the biggest runtime-deficit. Whenever a task is scheduled the leftmost task's avec_boosted value is decremented (The leftmost task is the task the original scheduler would take.) and the scheduled task's avec_boosted value is incremented. A positive value means that the task has been scheduled more often than it would have been with the old scheduler because of its characteristics. This is a very simple indicator of unfairness but therefore it always gives us a very significant result, what makes it very useful in comparison to average AVEC based mechanisms, which tends to give us inconclusive results if too many tasks are in the runqueue.

3.3.2 scheduling_position data structure

This value is an integer-array which tells us how a task has been ranked according to CFS' rbtree when it gets scheduled. It is only used for testing purposes. Its size is always AVEC_RANGE.

3.3.3 AVEC_RANGE parameter

The AVEC scheduling algorithm takes the n fairest tasks into account while searching for the next task to run. This n is called AVEC_RANGE. There is another similar value used in the migration code. Up to now we have used the same value for both, but that is not a must.

3.3.4 runtime_deficit_limit parameter

This threshold is the maximum entity_key a task may have to get scheduled (3.1).

3.3.5 soft_protection_threshold parameter

The distance between a task and the average task (a non-existing task with a AVEC which is absolutely average) must exceed this threshold in order to replace a actually fairer-to-schedule task (3.2).

3.4 The Avec Scheduling Algorithm

3.4.1 The Basic Algorithm

We let the standard algorithm search for the fairest task, but instead of taking the first one we take the AVEC_RANGE fairest tasks from the same group. Choosing the right value for AVEC_RANGE is some kind of trade-off. Higher values increase the probability to find a better matching task but they also increase the time needed to iterate through these tasks. They also increase the temporary imbalance among the tasks, because also tasks which are far behind in the fairness-sorted rbtree could get scheduled.

For each of them we calculate the distance to the average task and take the most dissimilar one. The distance function should be simple, but nevertheless give us clear results. The probably simplest approach would be just adding up the differences between the two AVEC's elements. This method would work for us, but it would have a drawback. If you have two tasks using the same functional units but stressing them unequally, their distance could be bigger than between two tasks not sharing any functional units, but stressing a smaller number of units. For our purpose less big differences are better than many smaller ones. That's why we use the AVEC elements to the power of two instead.

$$d \quad = \quad \sum_i |avec[i]^2 - avec_{avg}[i]^2| \tag{3.1}$$

For our dual core system the average AVEC consists of the arithmetic average of all running tasks' AVECs, but there are also other possibilities. Core2Quad CPUs consist of two separate dual core-dies in the same package. That means they share the same frontsidebus, but both parts have their own L2-Cache which would have been an

impact on our scheduling-decisions. It gets even more complicated, if you have more than one CPU. The Linux-Kernel can make a difference between the CPU-packages but not between the dies in the same package or at least it does not use this information while partitioning the CPU cores into scheduling-domains. The ACPI subsystem might deliver these information, but for this thesis we do not take it into account, because its impact should be small enough.

3.4.2 The Starvation Problem

Our modifications cause some problems with starvation which cannot occur in the original code. For example, if we have three tasks running, the first stressing only the fpu, the second stressing the integer unit and a third stressing both. Without any further protections the third one would never run because the other not-running task would always have a bigger distance. That is why we implement an explicit starvation-protection. Strictly speaking we have two ones and use them together. A soft one which tries to preserve fairness without affecting the AVEC scheduling and a hard one which enforces fairness whenever it gets too unfair.

While going through the sorted task list[1] the hard protection discards the following tasks when it comes to a task which's runtime deficit is too high. This task is then scheduled without any further considerations (Figure 3.1). That means if you set the deficit-limit to 0ms you would just get the unmodified scheduler's behaviour. Higher possible deficit-limits improve the scheduler's capability to optimize but decrease the fairness and therefore it can also have a bad impact on the reactiveness of interactive tasks.

```
FOR i = 0 TO (num_tasks - 1)
   IF ENTITY_KEY(i) > runtime_deficit_limit THEN
      SELECT_TASK(i)
      BREAK
```

Figure 3.1: hard protection

The soft protection is just a threshold. The algorithm tests the tasks in order of their fairness. A task's distance to the average AVEC must exceed the previously chosen one's distance by at least this threshold to replace it (Figure 3.2).

```
FOR i = 0 TO (num_tasks - 1)
   IF DISTANCE(i) > (DISTANCE(best) * soft_protection_threshold) THEN
      best = i
SELECT_TASK(best)
```

Figure 3.2: soft protection

3.4.3 Migration

For a better distribution of the tasks among the present cores we modified the migration mechanism to consider the task's AVECs. We took two different methods into consid-

[1] Remember: Tasks in CFS' rbtree are sorted by their fairness.

eration. The first approach uses a runqueue-AVEC which is calculated as the average AVEC of all tasks in the runqueue. We let the old migration mechanism decide from which runqueue a task should be moved, but we decide which task should be moved and where it should be moved to. This is done by comparing the tasks' AVECs with all runqueue-AVECs and choosing the most different one using the distance metric (3.1).

The second method uses the avec_boosted value to decide which task to move to another runqueue. It goes through the given runqueue and chooses the task which has been mostly displaced. We do not modify the algorithms for determining to which runqueue the task should be moved, because it is still the primary goal to distribute the load. That is why we let the default load-balancing-algorithm decide from which runqueue and to which runqueue a task should be moved. We just take care that it moves the most disadvantaged tasks from the given runqueue. Tests show that this works quite well, at least on our dual core machine. After each migration we have to reset the avec_boosted value or at least reduce its absolute value or the task would be migrated again because its value would be still very low.

The first method tells us not only which task should be migrated but also where it should be moved to. But some tests with different numbers of running tasks show that the runqueues' AVECs tend to get very similar with an increasing number of tasks, so that decisions are made on the basis of very small differences. The second method gives us a more significant metric but cannot be used to determine where a task should be moved to. Its advantage is its simplicity and its less impact on the general load-balancing. We decide to use the second method mainly because of the better significance of its results.

3.5 The Avec CPU Frequency/Voltage Scaling Algorithm

We use the cpufreq subsystem which comes with the Linux kernel [4]. For our Core2 CPU we choose the ACPI cpufreq-driver instead of the speedstep-driver, which is marked as deprecated. The driver takes every positive integer value as valid frequency. If this is not a frequency the CPU supports the driver sets the nearest possible frequency. For our further calculations we take only the AVEC values which are frequency-sensitive (all but bus/mem) into account. These values are taken from any running tasks' AVEC and their maximum defines the frequency. At first we calculate the frequency which would be set if only element i of the AVEC would be considered. Then we take the maximum of all these frequencies.

$$f(i) = avec[i] * f_{max}/avec_{max} \tag{3.2}$$

$$f_{max} = \max_i (f(i)) \tag{3.3}$$

3.6 Restrictions

For best results we would have to count any event at any time, but the CPU has only three fixed function pmcs and only two freely programmable counters so that we must use time-multiplexing to count more than two/five events. That means we change the events the counters count from time to time (We call such a set of counter-configurations a "rotation"). Reprogramming the pmcs is very time-consuming so that we have decided to "rotate" only each 100th scheduling pass. Most tasks do not change their requirements or at least not in a short period of time so that the impact of time-multiplexing on the resulting AVEC should be low enough.

Another problem is that you cannot count all events needed directly so that you must calculate some events on the basis of other countable events [8] which could be possibly on different rotations which makes the result even more inaccurate. But again it is not a problem as long as the requirements do not change too fast in terms of a small number of timeslices. For the calculation of the integer value we need the MMX/SSE and the Load/Store value, but there are instructions which are counted for both values so that we get a too low integer value whenever MMX/SSE is used.

For performance reasons we never reset the performance counters so that the they will overflow after a while but we do not treat this specially but let it just happen. This problem occurs very rarely (once in some years) and it is temporarily very limited (a few timeslices). In the worst case two very similar tasks are scheduled at the same time which leads to many stall-cycles during these timeslices.

With the integration of the new "Completely Fair Scheduler (CFS)" many new features were added to the scheduler infrastructure like scheduling classes and scheduling groups. For the implementation of AVEC scheduling we need access to the low level datastructures (the runqueue or a equivalent datastructure). That makes it impossible to implement the AVEC parts in the global schedule function, because the current CFS implementation just ignores some parameters from the main scheduling function. So we have decided to integrate the AVEC routines directly into the CFS scheduling function, because it is the standard class for all non-realtime tasks and therefore sufficient to test if AVEC scheduling works. But that also means that tasks which are in a different scheduling classes ignore the AVECs [2]

The AVEC frequency and voltage scaling was implemented as an usual governor for the ACPI module. It is designed to be callable either from scheduler or from a separate kernel-thread. In the current version calling it from scheduler does not work because of some issues of the ACPI driver when called from the scheduler. Switching the frequency/voltage directly should solve this problem, but we run out of time so that we do not implement this but use a kernel thread to set the frequency instead. This solution has of course a very bad impact on the performance. That is why we run only functional, but no performance tests while AVEC frequency/voltage scaling is active.

The current implementation works mostly with static parameters and sizes of datastructures which are sufficient for test-runs because we can adjust them and then recompile it for the certain test. For regular use this of course not an option. In our current version do not implement algorithms to adjust these parameters automatically, but due to the fact that they mostly depend only on the number of tasks it should be not much a problem to implement them later.

[2] The latest vanilla kernel only includes CFS- and RT-class.

Chapter 4

Experimental Results

4.1 Functional tests

4.1.1 Testing the Scheduling Algorithm

For our first experiment we use the burn-testprograms consisting of burnP5, burnP6, burnK6, burnK7 which mainly stresses the fpu, but also partly the branch-prediction units. burnMMX stresses the MMX-Unit and burnBX the memory subsystem. We let them run concurrently for about 10 minutes and then take the AVECs, avec_boosted values and scheduling positions (Table 4.1 and 4.2).

Task	Instr	Cycles	L2	Bus	MMX	SSE	x87	BR	Integer
burnP5	6822	10000	0	178	0	0	3208	201	202
burnP6	8559	10000	0	11	0	0	2443	1833	0
burnK6	8514	10000	68	0	0	0	2444	1833	0
burnK7	9096	10000	0	110	0	0	2139	1070	1607
burnMMX	7891	9999	10258	24	4925	0	0	1316	0
burnBX	2	9999	10386	11187	0	0	0	0	1

Table 4.1: Example: AVEC

Task	avec_boosted	pos1	pos2	pos3
burnP5	-4778	29769	2138	760
burnP6	-11925	32358	298	38
burnK6	-9936	32080	426	36
burnK7	-9980	31707	812	350
burnMMX	13004	15724	4089	12696
burnBX	25045	7458	4555	20747

Table 4.2: Example: avec_boosted value and scheduling positions

4.1.2 Testing the Migration Algorithm

Our first test is quite simple. We start ten instances of burnP6, burnBX and burnMMX, let them run some time and then look to which core they have been assigned to. The results of this test are quite similar with AVEC-kernel and standard Vanilla-kernel, with both kernels delivering an almost perfect distribution. Repeating the test with changed sequences, in which the tasks are started, does not affect the results significantly.

Our second test should make it harder for the kernels. We used the same test, but pinned all tasks to the first core. After that we changed the tasks affinity so that they can run on both cores again. This test shows a different behaviour; with a starting sequence such as P6,BX,MMX,P6,BX,MMX,... the results are similar to the first test's ones. The starting sequence P6,...,P6,BX,...,BX,MMX,...,MMX shows a completely different behaviour. The Vanilla-kernel moves all burnMMX tasks to one core and all burnP6 tasks to the other, while half of the burnBX tasks were moved to the second core and the other half stay on the first one. The AVEC-kernel delivers a different result which is better but far from perfect. We repeat the test three times and take the average distribution.

Task	Vanilla # tasks/core	AVEC # tasks/core
burnBX	4,33/5,67	5,33/4,67
burnMMX	0,67/9,33	6,33/3,67
burnP6	9,67/0,33	3,33/6,67

Table 4.3: Test: Migration

The distribution delivered by the Vanilla-kernel is not surprising because it always migrates the first movable task. On the other side the results of the AVEC-kernel is not as good as expected. The reason is most likely the scheduling behaviour and therefore the the avec_boosted value. At the moment we would have to adjust the static parameters for the AVEC subsystem (e.g. thresholds, number of considered tasks) according to the environment (e.g. number and sort of tasks) to get better results. In later versions we could probably replace some of these static data structures with dynamic ones so that the system becomes more adaptive.

4.2 Performance tests

4.2.1 Single Task

For each of these performance tests we have written a small shell-script and have measured the runtime with "time" command. We run each program three times and take the average runtime. The biggest deviation from the average was less than 3% so that we do not list the results for the single runs. The three tests we used here are prime, a simple prime number tester, kernel-build which compiles the AVEC Linux kernel with two threads (-j2) and mbw, a synthetic memory benchmark (used options: 7000 loops, fixed-block-read, 512MB/loop).

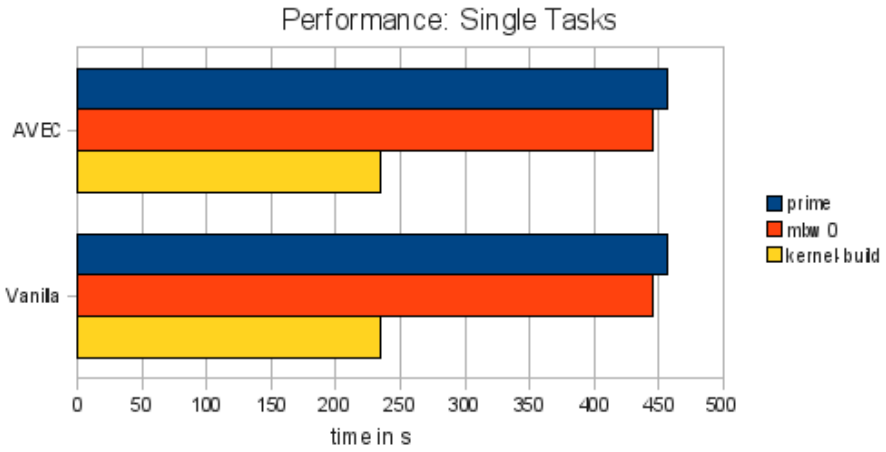

Figure 4.1: Single Tasks

Kernel	prime	kernel-build	mbw0
AVEC	7m56.2s	3m54.9s	7m25.5s
Vanilla	7m56.1s	3m54.8s	7m24.8s

Table 4.4: Single Tasks

4.2.2 Multiple Tasks

This tests consists of a shell-script which starts 2x prime, 1x kernel-build, 2x mbw0 at the same time. We run this test three times.

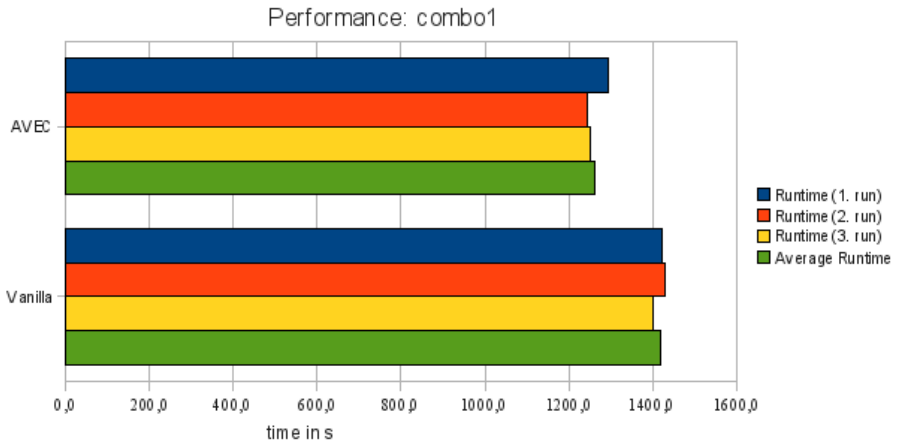

Figure 4.2: Multiple Tasks

Kernel	1. Run	2. Run	3. Run	Average
AVEC	21m34.020s	20m44.431s	20m50.232s	21m02.894s
Vanilla	23m43.726s	23m47.670s	23m21.588s	23m37.661s

Table 4.5: Multiple Tasks

4.2.3 Results

The results show what we have exspected. The Vanilla kernel is a little bit faster with one or two threads running. This can be explained with the overhead from the AVEC subsystem. On the other side, the AVEC kernel is significantly faster when it comes to multiple threads running. We tested it with six running threads which fits to the AVEC_RANGE of ten we have chosen as default.

Chapter 5

Conclusion

5.1 Conclusion

In retrospect the most difficult part has been the adaption of the AVEC subsystem for the CFS. The CFS works according to a simple principle, fairness. But it is this simplicity which makes it hard to modify the scheduler without disturbing the fairness, so that most of our modifications to the scheduler increases unfairness. That is why we let as many parts of the scheduler as possible untouched. Although you can definitively notice the unfairness introduced by our modifications. Watching the distribution of cpu-time with top shows that the modified scheduler produces some big spikes from time to time whereas the original one does not. This is caused be another mechanism integrated in the CFS. The CFS determines the length of a timeslice by the number of running tasks, but also by the tasks runtime-deficit, which can be significantly increased due to task-reordering. We can limit this behaviour as described (3.4.2), but that also limits the effect the implementation can have in terms of reducing hotspots and increasing performance. Generally speaking the CFS is not the best starting point for an extension like the AVECs, but due to the fact that CFS is the best scheduler for Linux at the moment so we have to live with it. Perhaps another scheduling algorithm, which is in the best case a priori designed for the use of AVEC would be a better solution and with the introduction of scheduling classes such a scheduler could be easily added.

The performance tests shows that this kind of scheduling surely have some potential. That hotspot-reduction through AVEC-Scheduling works has been shown in earlier papers. How good it works in combination with the aim of increasing performance is still to test. Using AVEC-based frequency/voltage scaling together with AVEC-scheduling is hard to combine, because most of today's CPUs allow only a single frequency for all cores, so that at first glance scheduling similar tasks in terms of needed CPU frequency would be the best solution. Combining this with the aim of reducing hotspots would be possible as long as there are enough tasks which need the same CPU frequency but stress different functional units. But this has a bad impact on the performance due to the fact that memory-intensive tasks would be scheduled together and the memory is already a bottleneck for most architectures [12]. The energy you would save by letting them run with lower frequency and voltage is overcompensated by the energy you loose because of a longer runtime, not to mention the worse performance. The CPU companies could solve this problem by supporting different frequencies/voltage for every single core.

5.2 Outlook

We have implemented AVECs to improve scheduling and frequency and voltage scaling. But there also other scopes for what you can use AVECs. An example, for what it could be possibly used for in the future, are power and clock gating. These mechanism are used to reduce power-consumption by switching off the power supply or at least the clock-signal for functional units which are temporary not in use. CPU makers use some sort of heuristics to determine when it is the best time to switch off/on each functional unit. But it is the same problem as with frequency-scaling. The CPU cannot know which functional unit the next task will need, because it cannot know which task that would be. On the other hand the scheduler obviously has this information. That is why the scheduler could iniate such a switch off/on even while switching the task. Because of the next's task AVEC the scheduler would perfectly know which units can be deactivated and which must be activated again. The present problem is that most (or even all?) CPU makers do not support software-controlled power and clock gating yet.

Another CPU feature which we do not include in our thoughts is multithreading (Intel calls it hyperthreading or just HT), because the Core2 architecture lacks this feature. Intel's older NetBurst architecture (e.g. Pentium 4 CPUs) and their new architecture Nehalem (e.g. Core i7 CPUs) supports hyperthreading so that it is quite interesting to think about how good AVEC-Scheduling would work on such a architecture. Multithreading is a technique to improve the efficiency of superscalar CPUs [14]. Superscalar cores have multiple functional units (FU) which can not always be fully utilised because of data dependencies in the code. A CPU with multithreading support provides virtual CPUs. The instructions for these virtual CPUs are executed in the same pipeline in the same time. The dispatcher can now choose between instructions from two different threads which are per se independant. This leads to a higher utilisation of the CPU's FUs and therefore to a improved performance due to a higher degree of parallelism. In some cases, if very similar threads are scheduled on the virtual CPUs, there will be no benefit because they want to use the same FUs. The AVEC-scheduler knows which FUs the tasks need and so it can schedule tasks on the virtual CPUs which do not interfere. It is the same mechanism which is used for scheduling on multicore CPUs, but for multithreading it should work even better because in this case all FUs are shared and not only the L2 cache, bus and memory.

Bibliography

[1] Cfs scheduler. In *Linux Kernel Documentation*.

[2] Frank Bellosa. Power management lecture, 2007. Karlsruhe, Germany, 2007.

[3] Daniel P. Bovet and Marco Cesati. In *Understanding the Linux Kernel, 3rd Edition*.

[4] Dominik Brodowski. Cpu frequency and voltage scaling code in the linux(tm) kernel. In *Linux Kernel Documentation*.

[5] Intel Corp. Volume 2a: Instruction set reference, a-m. In *Intel 64 and IA-32 Architectures Software Developers Manual*, April 2008.

[6] Intel Corp. Volume 2b: Instruction set reference, n-z. In *Intel 64 and IA-32 Architectures Software Developers Manual*, April 2008.

[7] Intel Corp. Volume 3b: System programming guide, part 2. In *Intel 64 and IA-32 Architectures Software Developers Manual*, February 2008.

[8] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. September 2003.

[9] Avinesh Kumar. Multiprocessing with the completely fair scheduler: Introducing the cfs for linux. In *IBM developerWorks*, Pune, India, January 8 2008.

[10] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. In *First ACM SIGOPS EuroSys Conference*, Leuven, Belgium, April 18–21 2006.

[11] Andreas Merkel and Frank Bellosa. Power management lab, 2007. Karlsruhe, Germany, 2007.

[12] Andreas Merkel and Frank Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *In Proceedings of the Workshop on Power Aware Computing and Systems (HotPower'08)*, San Diego, CA, December 7 2008.

[13] Andreas Merkel and Frank Bellosa. Task activity vectors: A new metric for temperature-aware scheduling. In *Third ACM SIGOPS EuroSys Conference*, Glasgow, Scotland, March 31–April 4 2008.

[14] Wikipedia. Superscalar. In *http://en.wikipedia.org/wiki/Superscalar*.