Event–Driven Thermal Management in SMP Systems

Andreas Merkel   Frank Bellosa
University of Karlsruhe
System Architecture Group
76128 Karlsruhe, Germany
{merkela, bellosa}@ira.uka.de

Andreas Weissel
University of Erlangen
Department of Computer Sciences 4
91058 Erlangen, Germany
weissel@cs.fau.de

Abstract

Actions usually taken to prevent processors from overheating, such as decreasing the frequency or stopping the execution flow, also degrade performance. Multiprocessor systems, however, offer the possibility of moving the task which caused a CPU to overheat away to some other, cooler CPU, so throttling becomes only a last resort taken if all of a system's processors are hot. Additionally, the different energy characteristics that different tasks are showing can be exploited and hot tasks as well as cool tasks can be distributed evenly among all CPUs.

This work presents a mechanism for determining the energy characteristics of tasks by means of event monitoring counters, and an energy–aware scheduling policy, which strives to assign tasks to CPUs in a way that avoids overheating individual CPUs. We implemented energy–aware scheduling for the Linux kernel. Evaluations show that the overhead incurred by additional task migrations is negligible compared to the benefit of avoiding throttling.

1. Introduction

With increasing clock speed and circuit density, power dissipation has become an issue in todays high-performance microprocessors. Currently, cooling facilities are designed for the worst case, i.e., a situation in which the CPUs are executing tasks that produce a maximum of heat.

However, most ordinary tasks do not reach this maximum. Therefore, the alternative is to design cooling facilities for a more moderate maximum power, and to throttle a processor if it executes tasks which exceed this power and becomes too hot. In multiprocessor systems, such situations can be avoided if the right decisions concerning which task to run on which CPU are taken.

In an operating system, the scheduler is the component responsible for deciding which task runs on which CPU. To get maximum performance out of a multiprocessor system under constraints imposed by the limited ability of each processor to dissipate energy without overheating, the scheduler has to know how much energy each task is consuming and how much energy can safely be dissipated on each processor per time unit, so it is able to make the right scheduling decisions and assign tasks to CPUs appropriately; it has to be *energy–aware*.

Schedulers found in contemporary operating systems try to maximize the throughput and the responsiveness perceived by the user. To make them energy–aware, they have to be extended to take the energy criterion into account, without neglecting their conventional criteria.

Therefore, we identify the following prerequisites: Firstly, we need a mechanism for determining the energy characteristics of a task, which means, how much energy a task is currently consuming and is expected to consume in the future. Secondly, we need a policy for deciding which task shall run on which CPU respecting the criteria of throughput, responsiveness, and energy.

Event monitoring counters included in modern microprocessors can be used to estimate the energy a processor spent during a certain period of time [3]. We use these counters to create task energy profiles describing the energy characteristics of the individual tasks. Our energy–aware scheduling policy is based on these profiles and strives to distribute energy consumption evenly among all CPUs of a system in order to minimize the need for throttling. We implemented this policy for the Linux kernel and integrated it with Linux's load balancer.

The rest of the paper is structured as follows: Section 2 reviews related work. Section 3 describes how task energy profiles are derived from event monitoring counter values. Section 4 presents the energy–aware scheduling policy. Section 5 briefly describes the changes done to Linux in order to integrate both the task energy estimator and the energy–aware scheduling policy. Section 6 shows some evaluations which have been done using the Linux implementation. Section 7 discusses directions for future work and the limitations of our approach. Section 8 concludes.

2. Related Work

2.1. Online Energy Estimation

Online energy estimation of the processor as a whole or of each individual task by means of event monitoring counters has been utilized in different ways to estimate and to control the chip temperature of the microprocessor.

For processors supporting frequency and voltage scaling, the energy characteristics of a task can be used to determine the optimal frequency at which the task should be run in order to reduce energy consumption without decreasing performance substantially [13]. However, frequency and voltage scaling is not available on most of todays high performance processors used in multiprocessor server machines.

The energy estimation of the processor as a whole can be used as input for a thermal model of the processor and its cooling facility [3]. This alleviates the need for reading the thermal diode to determine the current temperature, which is a time consuming operation. Additionally, the thermal model can be used to calculate the amount of energy that may be dissipated during a certain period of time without overheating the processor. This information is of vital interest for an energy–aware scheduler.

Combining compact models [7] of the processor with energy estimation from event monitoring counters allows the estimation of multiple temperature values for the different functional units on a processor chip [9]. However, considering more than one temperature value per chip for scheduling is beyond the scope of this work.

Macromodels [12] are another method for online energy estimation. A macromodel relates the energy consumption of a function or a block of code to various parameters that can either be observed or calculated from a high–level programming language description. However, the construction of macromodels is done off–line, so the applications in question must be known and analyzed in advance.

2.2. Energy–Aware Scheduling

The energy spent by individual tasks can be used to influence the decisions of an operating system's scheduler. Up to now, most of the work on energy–aware scheduling did not take multiple processors into account.

The abstraction of resource containers [1] allows the management of energy as a first class resource in distributed systems [14]. Only tasks whose corresponding energy container is non–empty may be chosen by the scheduler. Since this work's focus is not on choosing which tasks are allowed to be scheduled, but rather on where they are scheduled, the two approaches are fully compatible and could easily be combined.

The principle of moving computation away when temperature gets too high is part of the *Heat–and–Run* scheduling policy developed for simultaneously multithreaded (SMT) chip multiprocessors (CMP) [5]. Heat–and–Run characterizes tasks by their IPC value (instructions per cycle) and uses this characterization to co–schedule tasks on SMT sibling processors in a way that produces maximum heat. When a certain temperature limit is reached, tasks are migrated to another core on the CMP. However, Heat–and–Run was not implemented on real hardware, but uses the *Wattch* [4] architectural–level simulator.

Another approach consists in including spare resources like register files, ALUs, or issue queues on the processor chip and migrating computation to one of those spare resources when the original resource reaches a critical temperature [6, 11]. Our approach works on a more coarse grained level, moving computation not within a chip but between chips.

3. Task Energy Estimation

The decisions of an energy–aware scheduler are based on the energy characteristics of individual tasks, i.e. how much energy a task is currently consuming per time unit, and is expected to be consuming in the near future. This section describes a mechanism which allows the operating system to create an *energy profile* for each of the tasks it manages. This profile is an estimation for the energy the task will consume if it is allowed to run on a CPU for one timeslice.

3.1. The Need for Online Energy Estimation

On modern microprocessors, the energy characteristics of two tasks can differ substantially, depending on the type of instructions executed [3, 8]. An analysis of the processor's power consumption while running a particular task shows that power consumption is fairly static most of the time, but exhibits changes as the task experiences different phases of execution, e.g. runs different algorithms successively [2]. The sequence and the duration of these phases depends on the task's input data. Therefore, the energy characteristic of a task cannot be known in advance and thus cannot be determined by off–line analysis.

Energy consumption caused by tasks results in increased processor temperature. With the energy consumption known, it is possible to estimate temperature using a thermal model. Conversely, with temperature known, it is possible to estimate energy consumption — but not at the granularity in terms of time needed for attributing this energy consumption to distinct tasks. In general purpose operating systems, the length of a timeslice ranges between 10 and 100 milliseconds. Because of the huge ther-

mal capacitance of the processor's heat sink and the low resolution of the thermal diodes found in contemporary systems, the amount of energy dissipated during one timeslice is by orders of magnitudes too small for the diode to register a change in temperature. Additionally, reading the thermal diode has significant overhead (several milliseconds for reading the thermal diode via the system management bus [3]).

However, to characterize individual tasks, energy consumption must be known at timeslice granularity, since the CPU might be executing a different task each timeslice. As a solution, we apply online estimation of a task's energy consumption. Event monitoring counters found in contemporary processors like the Pentium 4 allow the estimation of the energy consumed during a period of time as short as one timeslice.

3.2. Transforming Events to Energy

We estimate the energy a processor spends by counting certain processor–internal events. We associate a fixed amount of energy with each event, and calculate the energy spent during a period of time using a linear combination of the counter values. This method for estimating energy has been implemented for the Linux kernel and yields an estimation error of less than 10% for real–world applications [3].

We determine the energy a task spends during one timeslice by reading the event counters at the beginning and at the end of the timeslice, calculating the difference for each event and weighting it with the corresponding amount of energy.

3.3. Energy Profiles

To make the optimal decision regarding on which CPU to run a task for the next time, the scheduler would have to know how much energy this task is going to consume during its next timeslice. This is not possible, since input data influence the behavior of tasks in a non–deterministic way. However, tasks show phases of different but static power consumption most of the time. Therefore, the energy a task consumed the last time it was executed is a good guess for the energy that the task will consume the next time it is allowed to run.

Considering the consequences a change in a task's energy profile might have, i.e. the migration of the task to another CPU (see section 4), we should avoid changing a task's energy profile too often and too drastically. Therefore, we do not only take the energy spent during the last timeslice into account, but use an exponential average of the task's past energy consumption. As a result, short term changes in a task's behavior do not cause the task's energy profile to change significantly, whereas a permanent change is reflected by the energy profile after an appropriate time.

The exponential averaging algorithm is intended for calculating the average of a value which is sampled over constant periods of time. Some operating systems, like Linux, give longer timeslices to tasks with higher priorities. Even if subsequent timeslices given to a task are of equal length, the actual time a task executes until the next one is scheduled may still differ from the duration of a timeslice. A task may block any time or be deprived of the CPU by a higher priority task in preemptively scheduled systems.

There are two solutions to this problem: Firstly, we can shorten the interval for calculating the exponential average, so the average is recalculated not only at the end of each timeslice but, for instance, on every timer tick. This way, the energy profile of a task is up to date even if it stops executing in the middle of a timeslice. Secondly, we can extend the exponential averaging algorithm to support variable periods, so we can calculate the exponential average at any time a task stops executing. We chose the latter approach, since it incurs less overhead (the exponential average must be recalculated less often) and is more flexible (a task might stop executing even between two timer ticks).

The conventional exponential average x_i for sampling period i is calculated by weighting the current sample value v_i (in our case, this is the energy spent during the sampling period) with a weight p. The exponential average of the previous sampling period x_{i-1} is weighted with $(1-p)$:

$$x_i = p \cdot v_i + (1-p) \cdot x_{i-1} \qquad (1)$$

Instead of using a constant weight p, we use a flexible weight which depends on the length of the sampling period. If the sampling period is shorter than a standard timeslice, we give the past a bigger weight, which compensates for calculating the exponential average more frequently. (The past values are weighted down with every iteration.) Vice versa, if the sampling period is longer than a standard timeslice, we give the past a smaller weight, because with longer timeslices the average is calculated less frequently.

4. Energy–Aware Scheduling

This section deals with the energy–aware scheduling policy and its implementation. We distinguish two cases: In situations where there is only one or less runnable task per CPU in the system, we use a technique called *hot task migration* to move tasks away from processors which threaten to overheat. In situations with more than one task per CPU, we employ *energy balancing*, a form of load balancing which takes the tasks' energy profiles into account and tries to distribute energy consumption evenly among all CPUs.

4.1. Scheduling in SMP systems

In SMP systems, each task is allowed to run on every CPU. However, there are reasons to avoid moving a task from one CPU to another one, unless this is really necessary. After a task has been running on a CPU for some time, it has warmed up the CPU's cache. Therefore, it is common practice in today's multiprocessor operating systems to distribute the set of all runnable tasks in a system among the system's CPUs. A subset of the tasks is assigned to each CPU and each CPU runs a scheduler of its own which chooses the task to be run next on the CPU out of this subset.

For each CPU, the scheduler must organize the corresponding subset of tasks in some kind of data structure, so it can keep track of these tasks. In Linux, this data structure is called the *runqueue*. Since every task belongs to one runqueue only, it is always executed on the same CPU unless it is transfered from one runqueue to another one.

The more tasks a runqueue consists of, the smaller is the share of CPU time each task gets. Since general purpose operating systems strive to provide fairness to their tasks, all runqueues should be of equal length. If we take SMP scheduling in its simplest form, equalizing the runqueue length (commonly referred to as *load balancing*) is the only reason why tasks should be moved between runqueues.

4.2. Scheduler Domains

In NUMA (non uniform memory access) systems, there is one more reason why tasks should be kept local to one CPU. If a task is moved to some CPU residing on another node, the memory this task is referencing must either be transfered to the new node, or the task has to do inter–node accesses. Both possibilities incur performance penalties. Therefore, if load balancing can be done between CPUs belonging to the same node, this should be preferred to load balancing between CPUs belonging to different nodes. Similarly, in SMT systems, load balancing should preferably be done between sibling CPUs, because they share the same cache.

To make optimal load balancing decisions, the scheduler has to know about the CPU topology of the system, i.e. who is whose sibling and who shares the same node with whom. Linux introduces an abstraction called *scheduler domains* to represent this topology [10].

A scheduler domain consists of two or more *CPU groups*. A CPU group is a set of CPUs. Load balancing in a scheduler domain is done between the domain's groups. This means the goal of load balancing is to have the same average runqueue length for all CPU groups. If group A has a greater average runqueue length than group B, tasks have to be migrated from some CPU in group A to some CPU in group B. To represent a system's topology, scheduler domains are stacked in a hierarchical fashion. The higher the level in this domain hierarchy, the costlier are the balancing operations in a domain. Therefore, load balancing is done on the lowest level possible.

An example: In an eight–way SMP system consisting of two NUMA nodes and with each processor being two–way simultaneously multithreaded (16 logical CPUs in total), there are three levels in the domain hierarchy: The domains on the lowest level each span one physical CPU and possess two groups consisting of one logical CPU. The domains on the second level span one node each, with each group consisting of two logical processors residing on the same physical CPU. Finally, the top level domain consists of two groups, and each group consists of eight logical CPUs whose corresponding physical CPUs all reside on one node.

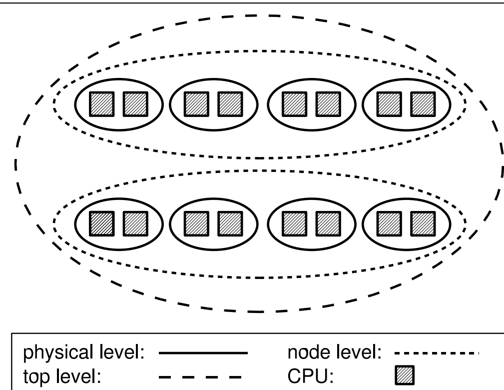

Figure 1. An example for scheduler domains

4.3. Objectives

What is true for load balancing also applies to *energy balancing*, i.e. migrating tasks for energy reasons: Moving tasks from one CPU to another should only be done when really necessary and within the lowest domain possible. However, an energy–aware scheduler may find reasons for moving tasks different from unequal runqueue lengths: For instance, hot tasks might have to leave hot processors, so these processors do not become even hotter and have to be throttled. If this is not done exceedingly often, the penalty incurred by moving tasks is redeemed by having all processors running at full speed.

Therefore, the main objective of energy–aware scheduling is: No processor should become so hot it has to be throttled. However, this can be achieved in different ways. The probably easiest solution is to wait until it is too late: An

energy–hungry task is allowed to run on a processor until the processor overheats and is then evacuated to some other processor. On systems with only few runnable tasks (one task or less per runqueue), and with a slight modification, this is even the best strategy: We do not wait exactly until it is too late, but only until it is nearly too late and move a hot task away from a processor slightly before the processor gets so hot it has to be throttled. The hot task can continue to run on a cooler processor which has either been idle before, or has been running a cool task that can continue to run on the hot processor. Since tasks are moved at the last possible moment, this method minimizes the number of task movements.

On systems with two or more tasks per runqueue, the before mentioned solution is no longer the best. Imagine a system with two CPUs and one runqueue consisting of two hot tasks and the other consisting of two cool tasks. Overheating a CPU could be avoided by moving the two hot tasks to the cool CPU and thus exchanging them with the two cool tasks from time to time. However, it is better to combine the tasks in a way that a cool one and a hot one are running on each CPU. We can eliminate the need for moving tasks if we distribute the tasks in such a way that the average power is the same for all runqueues.

However, this is only true if all processors possess the same energy characteristics. Actually, one processor may be located closer to some cooling facility like a fan or an air inlet than another one and may thus be able to dissipate more energy per time unit without overheating. So not the average power, but rather the ratio between this average and the maximum power a processor is able to dissipate without overheating should be equalized. This way, we can keep all processors at the same temperature.

The two objectives of energy–aware scheduling can be summarized as follows:

- If possible, the tasks of a system should be distributed among the different runqueues in a way that keeps all CPUs on the same temperature level.
- If a CPU running a hot task will overheat in the near future, and there is a cooler CPU, the hot task should be transfered to that CPU.

4.4. Thermal Model

We use a thermal model of the processor chip and its heat sink to estimate chip temperature from power consumption. The error resulting from estimating energy and then estimating temperature based on the energy estimate is smaller than 1 degree for real–world applications.

We model chip temperature with an exponential function which depends on the thermal capacitance and thermal resistance of the chip and heat sink as well as on ambient temperature. The model is described in detail in [3].

To determine the coefficients of this exponential function, we started a task which produces a maximum of heat on a processor formerly idle, recorded the temperature values over time and fitted an exponential function to the experimental data.

4.5. Measures

We will now introduce some measures on which the energy–aware scheduler's decisions are based. First of all, the scheduler needs to know how much energy each CPU is currently consuming, because we want to balance energy consumption between the CPUs. Since the scheduling intervals are much shorter than the time it takes for the processor and the heat sink to warm up noticeably, we calculate the energy consumption of a CPU by averaging the energy values taken from the energy profiles of all tasks in the runqueue.

However, the *calculated consumption rate* obtained this way is only of limited accuracy. The timeslice length might be different for different tasks; higher priority tasks might get longer timeslices, as is the case in Linux. Additionally, the scheduler can never know whether all of the tasks in a runqueue will fully utilize their timeslice, or whether some of the tasks will block in the middle of a timeslice and will no longer be runnable. Hence the *calculated consumption rate* is only an approximation for the energy that is currently consumed by a CPU and will likely be consumed in the future.

Another shortcoming of the calculated consumption rate is that it only considers tasks which are currently members of the runqueue. This is insufficient for two reasons: Firstly, there may be tasks which are blocking and unblocking frequently, e.g. a web server. Since such tasks usually resume executing on the same CPU, they should also be considered when they are blocked and thus not members of the queue. Secondly, the energy dissipated by tasks which were executing on a CPU in the past is still partly stored in the processor chip and the heat sink. The scheduler must therefore not only consider current and future energy consumption, but also chip temperature, which is determined by past energy consumption.

To overcome these shortcomings, the scheduler bases its decisions on a second consumption rate as well, which is determined in an empirical way and is hence called *empirical consumption rate*. This empirical consumption rate mirrors the current CPU temperature. Whenever a task has used up its timeslice, is deprived of the CPU for some other reason, or releases it voluntarily, the scheduler looks at how much energy the task has consumed. It uses this value to calculate an exponential average similar to the task energy profiles, but considering any task running on a CPU instead of being task specific.

Since we determine this consumption rate in an empirical way, the rate only considers the tasks to the extent of their real runtime and is therefore more accurate. Since we calculate an exponential average, the empirical consumption rate also considers tasks which ran in the past but are no longer members of the runqueue.

The empirical consumption rate mirrors the energy which was dissipated by tasks in the past and is still stored in the processor and the heat sink. Therefore we fit the exponential average used to calculate this consumption rate to the exponential function of our thermal model. If a hot task has just been migrated away from a processor and the temperature of the processor as well as the empirical consumption rate are at their upper limits, the consumption rate should be halfway down to the consumption rate of a processor being idle just when the processor's temperature is. We calibrate the empirical consumption rate by choosing an appropriate weight p for the exponential average (see equation 1) which corresponds to the time constant of the exponential function from the thermal model.

However, we cannot do without the calculated consumption rate, which has one advantage over the empirical one: Since the calculated rate is the average of the energy profiles of all tasks in a runqueue, it immediately considers a task that is migrated to or started on a CPU, whereas the empirical rate does not consider such a task until it has used up its timeslice for the first time. This is important to avoid over–balancing, i.e. migrating too many tasks so one imbalance is replaced by another imbalance into the opposite direction.

As motivated in the preceding subsection, we do not necessarily want to balance the consumption rates, but rather their ratios with respect to a CPU–specific maximum consumption rate. Therefore, this maximum rate must be known to the scheduler, so it can calculate the ratios of both the exponential and the calculated consumption rate with respect to the maximum rate.

4.6. Energy Balancing

For scalability reasons, energy balancing uses a distributed algorithm similar to Linux's load balancing algorithm. The algorithm runs on every CPU, possibly in parallel, and works in two steps: During the first step, the algorithm searches for another queue to do balancing with. The second step consists of moving tasks between the two queues in order to resolve imbalances.

Only tasks which are currently not executing but waiting until it is their turn to get a timeslice of CPU time are transfered this way. This is called *passive balancing*.

As the balancing algorithm is executed on every CPU, balancing needs only be done in one direction: The Linux load balancer, e.g., only pulls in tasks from remote runqueues in order to resolve imbalances. If there is an imbalance which would require pushing out tasks, this imbalance is resolved when the load balancer runs on the remote CPU. Similarly, we do energy balancing only by pulling in "heat" from other runqueues (with some exceptions, as we will see later).

To avoid ping–pong effects (tasks being migrated back and forth), balancing should be done rather conservatively. This is where our two consumption rates described in the preceding subsection come into play: A remote queue is considered hotter than the local queue only if the minimum of both remote rates is bigger than the maximum of the local rates. Since the empirical consumption rate is only changing slowly, this creates a hysteresis effect, while the calculated rate, changing immediately, forbids to pull over an undue number of tasks. Similarly, when comparing ratios, we use the maximum of the local and the minimum of the remote ratios.

Energy balancing decisions must be consistent with the load balancing ones and vice versa. Otherwise, a task movement made for energy reasons might be undone again for load balancing reasons. Therefore, the energy balancer must always strive not to create load imbalances and the load balancer must strive not to create energy imbalances. Since load and energy balancing are intertwined (energy consumption is always bound to tasks), we decided to merge the existing Linux load balancing algorithm with energy balancing into one algorithm.

The following steps are executed for every scheduler domain a CPU doing balancing belongs to, beginning with the lowermost:

- First we search for the CPU group with the highest average energy ratio in order to do energy balancing.
- If the group containing the local CPU is found to be the one with the highest ratio, which is quite probable because we are comparing conservatively, there is no need to do energy balancing.
- Otherwise, we search for the queue with the highest ratio among the queues in the remote group and do balancing with this queue.
- We pull over tasks having energy profile values higher than the consumption rate of the remote queue in order to cool down the queue. However we do only pull such a number of tasks that
  - No load imbalance is created.
  - No energy imbalance in the wrong direction is created.

  Sometimes it might not be possible to pull over tasks without creating a load imbalance, e.g. if both queues

are of equal length. In this case, we exchange a low energy task from the local queue for a high energy task from the remote queue.

- Then we search for the most loaded CPU group in order to do load balancing.
- If the local CPU belongs to the most loaded group, there is no need to do load balancing.
- Else we search for the longest runqueue in the most loaded group.
- Again, we pull over tasks
  - Without creating a load imbalance in the opposite direction.
  - Without creating an energy imbalance.

  Note that, as opposed to energy balancing, load balancing can always be done without creating an energy imbalance: If the remote queue is hotter than the local one, we pull over a task which is hotter than the remote runqueue's consumption rate. If the remote queue is cooler, we pull over a task cooler than the remote runqueue's consumption rate.

4.7. Hot Task Migration

Passive balancing, as described above, works well if we want to equalize the energy ratios of CPUs running several tasks. However, if we are dealing with a runqueue which consists of one task only, passive balancing cannot be applied: We cannot transfer a task which is currently executed.

However, as described in subsection 4.3, a task must be moved to another CPU (assuming there is a suitable one) if it has caused the CPU it is running on nearly to overheat. So if a CPU's empirical consumption rate comes closer to the maximum consumption rate than a predefined threshold — which, due to the calibration of this rate to the CPU's thermal characteristics, is coterminous with the CPU nearly having reached its temperature limit — and the CPU's runqueue consists of one task only, we use *active migration* to transfer the task elsewhere. Since a task cannot be migrated while running, active migration has to be done by a special task, which is woken up in order to preempt the currently running task and to transfer it to its destination CPU. Again, if the destination CPU is already running a task, we transfer this task to the local CPU in return to avoid creating a load imbalance.

The destination CPU should be a CPU which is significantly cooler than the source CPU and which is either idle or running a low–energy task. But unless necessary, migrations across node boundaries should be avoided in NUMA systems. Therefore, we traverse the scheduler domain hierarchy similarly to energy balancing bottom–up, searching for the coolest runqueue within the domain. If the difference of the empirical consumption rates of this coolest queue and the local one is bigger than a predefined constant, we take this queue as the destination, else we continue searching one level higher in the domain hierarchy. If no suitable CPU is found after searching the top–level domain, all of the system's CPUs are hot and the hot task must remain on the hot CPU, which in turn will have to be throttled.

4.8. SMT Issues

Since the logical CPUs of a simultaneously multithreaded processor mainly use the same functional units on the same physical chip, there is no need to do energy balancing between them. We take care of this by means of the scheduler domain abstraction: The scheduler domains on the lowest level, which encompass all logical CPUs belonging to the same physical processor, are marked with a special flag, which tells the scheduler not to do energy balancing, so the energy balancing step is skipped for those domains. Load balancing, on the other hand, must still be done between sibling CPUs, but the energy restrictions for load balancing mentioned above do not apply.

Since energy balancing must be done between logical CPUs belonging to different physical processors, we still need the measures for energy–aware scheduling. Furthermore, because of the logical CPUs running independently of each other, we need them for every runqueue. Therefore, we divide the maximum energy consumption rate a physical processor can endure without overheating between all logical CPUs.

Note that due to energy balancing not being done between sibling CPUs, it may be that one logical CPU operates above the maximum consumption rate while another one operates below. However, on the next higher scheduling domain level, where energy balancing is done, all logical CPUs of one processor are collected in one group. Only the average of the group matters, so hot tasks from the over–hot logical CPU are not necessarily transfered to some other CPU.

Similarly, since not logical but only physical processors can overheat, we only migrate a hot task actively from a logical CPU belonging to a simultaneously multithreaded processor, if the sum of the empirical consumption rates of all logical CPUs belonging to a physical processor is greater than the allowed maximum rate for that processor. Again, we skip the lowermost level of the scheduler domain hierarchy when searching for a destination CPU, since migrating the hot task to a sibling CPU does not improve the situation.

5. Integration into the Linux Kernel

We did the following modifications to the Linux kernel to incorporate energy–aware scheduling:

- We integrated an energy estimation mechanism, which transforms event counter values to energy values.
- We extended the runqueue data structure to encompass the two consumption rate values as well as the allowed maximum rate.
- We enhanced the `task_struct` data structure, which Linux uses to describe tasks, by a field holding the task's energy profile.
- We implemented a mechanism for calculating the consumption rates and the task energy profiles from the energy values delivered by the estimator.
- We replaced Linux's load balancing algorithm with the combined energy–load balancing algorithm described in section 4.6.
- We added active migration for hot tasks.

All those modifications sum up to roughly 1800 lines of C–code.

6. Evaluation

For testing, we ran our modified Linux kernel on an 8–way Pentium 4 Xeon multiprocessor (2.2 GHz each processor) consisting of two NUMA nodes with four 2–way multithreaded processors each.

program	energy	description
bitcnts	$61W$	bit counting operations
memrw	$38W$	memory reads/writes
aluadd	$50W$	integer additions
pushpop	$47W$	stack push/pop
openssl	$42W - 57W$	OpenSSL benchmark
bzip2	$48W$	file compression

Table 1. Programs used for the tests

6.1. Energy Balancing

To test energy balancing, we set the energy limit of all CPUs to $60W$, so energy consumption should be balanced evenly across all CPUs.

We ran a mixed workload consisting of six different programs (see Table 1) and started each program three times, for a total of 18 running tasks. All tasks showed fairly static energy characteristics, with the exception of OpenSSL, which we ran in benchmark mode. Due to the different encryption and checksum algorithms benchmarked successively, the energy profile of OpenSSL varied between $42W$ and $57W$.

For comparison, we first ran the workload with energy balancing disabled. Figure 2 depicts the experimental consumption rates for the eight processors. Because of the exponential average used for calculating these rates, the rates rise exponentially first. This mirrors the exponential rise of the processors' temperatures. During the further course of the experiment, the curves diverge because of the different energy characteristics of the tasks running on each CPU. There were 44 task migrations during the 14 minutes the test was run, mainly because of interactive tasks waking up and thus creating load imbalances.

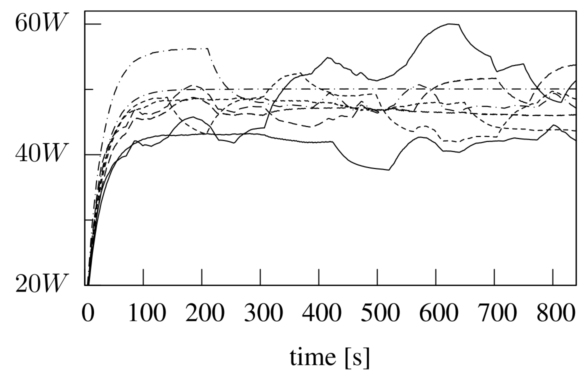

Figure 2. Energy balancing disabled

Figure 3 shows the energy consumption of the eight processors with energy balancing enabled. Although there is a variation in the overall energy consumption because of the non–static behavior of the OpenSSL benchmark, the width of the array of curves is limited. With the combined energy–load balancer, 193 task migrations happened, so there was roughly one migration every four seconds. Considering the total of 18 running tasks, each task was migrated on average every 78 seconds.

6.2. Hot Task Migration

For the next test,each physical processor was allowed to consume $40W$ at most, yielding a $20W$ limit for each logical CPU. We started a single instance of the bitcnts program, consuming about $60W$.

Since we have only one running task, the sibling of the logical CPU the task is running on is always idle. If the bitcnts task is started on or migrated to a CPU which was formerly idle, it takes approximately ten seconds for the sum of the experimental consumption rates of the two sibling CPUs (one idle, one executing bitcnts) to rise above $40W$.

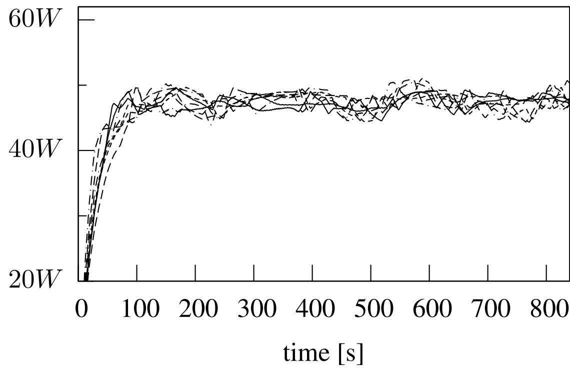

Figure 3. Energy balancing enabled

The bitcnts task is then migrated elsewhere by the hot task migration mechanism.

If bitcnts were executed on one processor all of the time, as is the case without energy–aware scheduling, this processor would have to be throttled 33% of the time to enforce the $40W$ limit, assuming that a processor is consuming no energy when throttled.

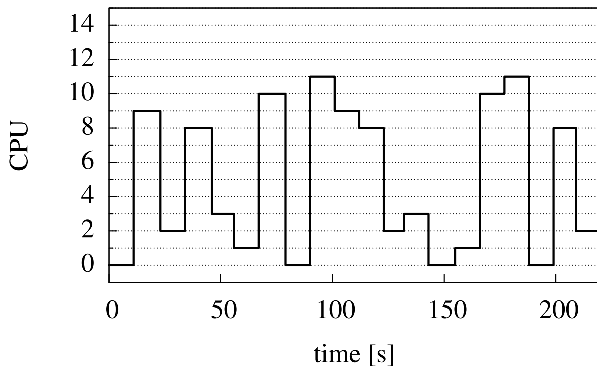

Figure 4. Hot task migration

Two more things are worth noting: Firstly, bitcnts is never migrated from one sibling to the other one, as you can see from Figure 4: The CPU IDs of two sibling CPUs differ in the most significant bit. Thus, CPU 0 is the sibling of CPU 8, CPU 1 is the sibling of CPU 9, and so forth.

Secondly, bitcnts is never migrated across the node boundary: CPUs 0 to 3 (with their siblings 8 to 11) reside on node 0, whereas CPUs 4 to 7 (with their siblings 12 to 15) reside on node 1. The bitcnts task visits the physical CPUs of node 0 nearly in round robin fashion, because the CPU least recently visited is always the coolest one. However, after bitcnts has taken one full turn, the CPU on which it executed first has cooled down enough to avoid inter–node migration.

6.3. Temperature Control

For this last test, we ran the workload consisting of the programs listed in Table 1 again. Without temperature control, the maximum processor temperature measured for this workload was 45°C. We employed a throttling mechanism to control the processors' temperatures: Whenever a CPU's experimental consumption rate rose above the value corresponding to a temperature of 38°C, we throttled the processor by executing the `hlt` instruction.

Again, we ran the test first with energy balancing disabled and then with energy balancing enabled. Table 2 shows the percentages of the time the CPUs were throttled for the two runs. The CPUs not shown in the table had to be throttled in neither of the test runs due to their good thermal characteristics. (Their temperature does not exceed 38°C even if the hottest task, bitcnts, is executed on them.)

CPU	energy balancing disabled	energy balancing enabled
0	51.5%	35.1%
3	54.1%	39.7%
4	10.8%	0.0%
8	61.1%	35.7%
11	54.7%	51.9%
12	11.0%	0.0%
average	15.2%	10.2%

Table 2. CPU throttling percentage

As expected, if energy balancing is enabled, the throttling percentage is lower for all CPUs (except for the ones that do not have to be throttled even with energy balancing disabled), because the balancing policy moves hot tasks to the processors with better thermal characteristics. The processors with poorer thermal characteristics, on the other hand, have to be throttled less often, because they are executing cooler tasks. The reduced throttling percentage results in shorter execution times for the test programs. The throughput (number of tasks finished per time unit) increased by 4.7% percent with energy–aware scheduling enabled.

6.4. Benefits

To assess the benefits of energy–aware scheduling, we must weigh the performance penalties incurred by additional task migrations against the performance boost gained

by avoiding the throttling of CPUs. We argue that the performance penalties are negligible compared to the performance boost.

If a task is migrated every ten seconds, it executes in the order of ten billion instructions between two migrations on a recent processor. Caches however, can be considered warm after executing some millions of instructions. This is by three orders of magnitude less, so the performance penalty is within the sub percent range. Throttling times, on the other hand, can be reduced by several percent by means of energy–aware scheduling.

7. Limitations and Future Work

The main limitation of energy–aware scheduling is that it is only applicable for workloads consisting of tasks with different energy characteristics. If all tasks possess the same characteristics, there is no need to do energy balancing, since energy is inherently balanced.

Currently, most processors are equipped with a single thermal diode. Throttling mechanisms are engaged by the system BIOS or some monitoring hardware when the temperature value reported by this diode exceeds a certain threshold. Since energy is dissipated at individual functional units of a processor, the temperature of the chip may be distributed non–uniformly, so decisions about throttling should be based on multiple temperature values. As a consequence, the goal of an energy–aware scheduling policy should be to keep the temperature of all functional units below the throttling threshold. Future work on energy–aware scheduling could incorporate a more elaborate thermal model featuring multiple temperatures.

8. Conclusions

We introduced a mechanism for estimating the energy consumption of a computer's microprocessor on a per–task level. We presented an energy–aware scheduling policy, which based on these energy profiles, strives to equalize the energy consumption with respect to a certain CPU–specific maximum consumption rate across all of a system's CPUs. In situations in which this is not possible due to a limited number of tasks in the system, the scheduler keeps hot tasks running on a CPU as long as possible and migrates them elsewhere if the processor threatens to overheat.

Evaluations show that the policy achieves its goal. Compared to the benefit of avoiding the throttling of CPUs whenever possible and the resulting performance boost, the overhead incurred by hot task migration or the task migrations required for energy balancing is negligible.

References

[1] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating System Design and Implementation OSDI'99*, Feb. 1999.

[2] F. Bellosa. The case for event-driven energy accounting. Technical Report TR-I4-01-07, University of Erlangen, Department of Computer Science, June 2001.

[3] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner. Event–driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*, Sept. 27 2003.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural–level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM Press, 2000.

[5] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. *SIGARCH Comput. Archit. News*, 32(5), 2004.

[6] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In *Proceedings of the International Symposium on Low Power Electronics and Design*, Aug. 2003.

[7] W. Huang, S. Ghosh, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Compact thermal modeling for temperature-aware design. In *Proceedings of the 41st ACM/IEEE Design Automation Conference (DAC)*, June 2004.

[8] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, Dec. 2003.

[9] K.-J. Lee and K. Skadron. Using performance counters for runtime temperature sensing in high–performance processors. In *Proceedings of the Workshop on High-Performance, Power–Aware Computing (HP-PAC)*, Apr. 2005.

[10] `linux/Documentation/sched-domains.txt`. Documentation shipped with the Linux source code.

[11] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA'03)*, June 2003.

[12] T. K. Tan, A. Raghunathan, G. Lakshminarayana, and N. K. Jha. High-level energy macro-modeling of embedded software. In *IEEE Transactions on Computer-Aided Design*, Sept. 2002.

[13] A. Weissel and F. Bellosa. Process cruise control — event–driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002)*, Oct. 8–11 2002.

[14] A. Weissel and F. Bellosa. Dynamic thermal management for distributed systems. In *Proceedings of the First Workshop on Temperature-Aware Computer Systems (TACS'04)*, June 2004.