View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by KITopen

Balancing Power Consumption in Multiprocessor Systems

Andreas Merkel
merkela@ira.uka.de

Frank Bellosa
bellosa@ira.uka.de

System Architecture Group
University of Karlsruhe
76128 Karlsruhe, Germany

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 that 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 scheduler can take advantage of the energy characteristics of individual tasks, and distribute hot tasks as well as cool tasks 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 that strives to assign tasks to CPUs in a way that avoids overheating individual CPUs. Our evaluations show that the benefit of avoiding throttling outweighs the overhead of additional task migrations, and that energy–aware scheduling in many cases increases the system's throughput.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*multiprocessing, scheduling*

General Terms

Management, Performance

Keywords

Energy–aware scheduling, energy estimation, event counters, task migration, thermal management

1. INTRODUCTION

With increasing clock speed and circuit density, power dissipation has become an issue in todays high–performance microprocessors. The arrival of multithreaded and multi–core architectures aggravates thermal problems, since concentrating more computational activity on ever smaller chip areas also means concentrating more heat, leading to increased chip temperature.

In the past, cooling infrastructures were designed for the worst case. For a microprocessor, this worst case means a situation in which the processor is constantly operating at its theoretical maximum power and thus dissipating a maximum of heat. However, most ordinary tasks do not cause the processor to consume this maximum power. Therefore, the alternative is to choose a more moderate thermal design power, and to throttle a processor if it becomes too hot after executing tasks that exceed this thermal design power. However, throttling means slowing down the processor, e.g., reducing its frequency or introducing halt cycles. Hence, throttling degrades the system's performance and should only be applied if really necessary.

The power consumption of recent processors depends strongly on the kind of instructions the processor executes and thus on the currently running task [8, 18]. *Hot tasks* cause high power consumptions and processor temperatures, whereas *cool tasks* consume less power, which results in lower processor temperatures. Hence, if the processors of a multiprocessor system are executing different tasks, they are likely to have different temperatures. This can lead to situations in which some *hot processors* have to be throttled, while other *cool processors* have lower temperatures. In such situations, we can avoid throttling if we take the right decisions concerning which task to run on which CPU. For example, we can combine hot tasks with cool tasks or migrate hot tasks to cool processors.

In an operating system, the scheduler is the component responsible for assigning tasks to CPUs. 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 take 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 criterion of energy 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 the 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 consumes during a certain period of time [6, 19]. We use these counters to create energy profiles describing the energy characteristics of individual

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'06, April 18–21, 2006, Leuven, Belgium.
Copyright 2006 ACM 1-59593-322-0/06/0004 ...$5.00.

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 reduce the need for throttling. We implemented this policy for the Linux kernel and integrated it with Linux's load balancer. In our experiments, energy–aware scheduling yields a 5% increase in throughput for a system with all CPUs busy by avoiding the throttling of processors.

The rest of the paper is structured as follows: Section 2 reviews related work. Section 3 describes how we derive task energy profiles from event monitoring counter values. Section 4 presents our energy–aware scheduling policy. Section 5 briefly describes the changes done to Linux in order to integrate both task energy estimation and energy–aware scheduling. Section 6 shows the results of evaluations we did with the Linux implementation under a number of workload scenarios. Section 7 discusses directions for future work and the limitations of our approach. Section 8 concludes.

2. RELATED WORK

2.1 Server Power Management

The main field for multiprocessors is in the area of server systems. Server systems offer many possibilities for leveraging their unique architecture and workload characteristics to reduce energy consumption and cooling costs by power management [22].

A multitude of other work on server power management addresses server systems offering special hardware support for processor power management, such as servers consisting of low–power processors [12], systems allowing frequency scaling for each individual processor [14], or systems composed of heterogeneous processors [20, 4].

In contrast to these, our approach does not rely on any hardware support for power management. The only prerequisite from the hardware side is the existence of event monitoring counters that are able to monitor hardware activity on the processor chip, which are already present in many of todays processors.

2.2 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 used in different ways to estimate and to control the chip temperature of the microprocessor.

Energy estimation of the processor as a whole can be used as input to a thermal model of the processor and its heat sink [8]. This alleviates the need for reading the thermal diode to determine the current temperature, which consumes a significant amount of time [8]. 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 [17] 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 [21]. However, considering more than one temperature value per chip for scheduling is beyond the scope of this work.

Macromodels [28] 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.

2.3 Energy–Aware Scheduling

An energy–aware scheduler bases its scheduling decisions on the energy characteristics of individual tasks. Up to now, most of the work on energy–aware scheduling did not take multiple processors into account. To our knowledge, no prior work has pursued the approach of doing multiprocessor scheduling based on the tasks' power consumptions.

If the operating system knows which tasks are responsible for a rise in processor temperature, it is able to keep the temperature below a certain limit by throttling only the hot tasks, instead of penalizing all tasks [24]. We argue that in multiprocessor systems, if there are cooler processors, migrating a hot task to such a processor is superior to throttling.

For processors supporting frequency and voltage scaling, CPU utilization and user–perceived interactive performance [13], or the energy characteristics of individual tasks [30], have been used to determine the optimal frequency at which the processor conserves power without decreasing performance substantially. Energy–aware real–time scheduling policies take advantage of slack time [23, 3] or of the energy characteristics of individual tasks [2] to decrease the processor's frequency if this does not lead to missing deadlines. However, frequency and voltage scaling are not available on most of todays high performance processors used in multiprocessor server machines.

The abstraction of resource containers [5] allows the management of energy as a first class resource in distributed systems [31]: The scheduler only chooses tasks whose corresponding energy container is non–empty. Ecosystem [32], another approach based upon resource containers, targets battery powered systems and offers a unified scheme for accounting power consumption caused by all system devices. Ecosystem aims at maximizing battery lifetime. The focus of this work, however, is not on reducing, but on distributing power consumption. We believe that these are different and — to a large degree — orthogonal aspects of power management, so that our proposed policy for balancing processor power consumption could be combined with any policy limiting overall power consumption.

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) [15]. Heat–and–Run characterizes tasks by their resource usage and, with this knowledge, co–schedules tasks on SMT sibling processors in a way that produces maximum heat, migrating tasks to another core when the temperature limit is reached. While the goal of Heat–and–Run is to seize the capacity of as many functional units on a core as possible before any unit overheats and causes the entire core to be paused, our policy leverages the low temperature of entire processors to reduce the stress on hot processors. Other work focused on SMT scheduling aims at attaining an optimal throughput by combining tasks on the logical CPUs of an SMT processor [9, 26, 11]. Similarly to our work, performance counter or similar values guide the search for a suitable combination of tasks.

Another approach consists in adding spare resources such as 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 [16, 25]. Our approach works on a more coarse grained level, moving computation not within a chip but between chips. Also, we make decisions at the operating system level rather than at the hardware level, which allows us to use knowledge about higher level abstractions such as tasks, which is not available to the hardware.

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 that 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 Need for Online Energy Estimation

On modern microprocessors, the energy characteristics of two tasks can differ substantially depending on the type of instructions executed [8, 18]. 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 [7]. The sequence and the duration of these phases depend on the task's input data. Also, because of cache and paging effects, the behavior of a task depends to some degree on the other tasks running in the system. For these reasons, the energy characteristics of a task cannot be known in advance and thus cannot be determined by off–line analysis.

The temperature of a processor is related to its energy consumption. With 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 to attribute 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 thermal 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 orders of magnitude 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 [8]).

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 the processor's power consumption, which allows to determine the amount of energy a processor consumes during a period of time as short as one timeslice.

3.2 Transforming Events to Energy

Modern processors, such as the Pentium 4, include special registers called *performance monitoring counters*, or more general, *event monitoring counters*. These counters are intended for delivering values useful for performance analysis and optimization, and are therefore able to count various processor–internal events. Since those events correspond to activities on the processor chip, event counters are suitable for estimating the energy the processor spends during a certain period of time.

Following the approach described in our previous work [8], we assume that the processor consumes a certain fixed amount of energy for each activity, and assign a weight to each event counter representing the amount of energy the processor consumes while performing the activity corresponding to that specific event. Based thereon, we estimate the energy consumption of the processor as a whole by choosing a set of n events that can be counted simultaneously, and by weighting each event with its corresponding amount of energy a_i. Thus, we determine the amount of energy the processor spends during a certain period of time by counting the number of events that occur during that period of time, and by calculating a linear combination of the counter values c_i:

$$E = \sum_{i=1}^{n} a_i \cdot c_i \qquad (1)$$

The weights a_i are calibrated by measuring the real energy consumption with a multimeter for several test applications, counting the events that occur during the test runs, and solving the resulting linear equations. 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 [8].

Following this approach, 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 the next time, the scheduler would have to know how much energy the 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. Since tasks show phases of different but static power consumption most of the time [7], 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. Table 1 substantiates this: For several tasks, we measured the processor's power consumption during the individual timeslices of each task's execution. We measured the power consumption during several hundreds of timeslices for each task, and compared the power consumption of successive timeslices. Although there can be significant changes in the power consumption of successive timeslices because of phase changes, these significant changes only occur rarely, and most of the time, the change in power consumption is small, resulting in a low average change.

The misprediction of a task's energy profile has no dramatic consequences, provided that it does not happen too frequently, since energy–aware scheduling is only a best–effort approach to minimize throttling. Yet, there is a difference between detecting a phase change some timeslices too late, as opposed to detecting a change when there is only a momentary spike in power consumption: A changed energy profile entails a reaction from side of the scheduler (see Section 4), hence detecting a false change can provoke an unnecessary task migration, whereas delaying a change reduces the number of task migrations.

To avoid changing a task's energy profile too often and too drastically, instead of only taking into account the energy spent during the last timeslice, we use an exponentially weighted moving average (exponential average, for short) of the task's past energy consumption to predict the energy profile. 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 in the energy profile after an appropriate time.

Table 1: Change in power consumption during successive timeslices

program	maximum	average
bash	19.0%	2.05%
bzip2	88.8%	5.45%
grep	84.3%	1.06%
sshd	18.3%	1.38%
openssl	63.2%	2.48%

The exponential averaging algorithm is intended for calculating the average of a value 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: The first solution is to 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 the task stops executing in the middle of a timeslice. The second solution is to 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 exponential average $\overline{x}_i$ for sampling period i is calculated by weighting the current sample value x_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 $\overline{x}_{i-1}$ is weighted with $(1-p)$:

$$\overline{x}_i = p \cdot x_i + (1-p) \cdot \overline{x}_{i-1} \quad (2)$$

Instead of using a constant weight p, we use a variable weight that 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.) Conversely, 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

As we shall see in Section 6, migrating tasks based on their energy profile is apt to avoid high temperatures of individual processors and thus the performance penalties of throttling. However, the migration of tasks introduces performance penalties, too, since task migrations break processor affinity, which we will discuss in Section 4.1. Energy–aware scheduling is beneficial if the penalty incurred by migrating tasks is compensated by having all processors running at full speed. Hence, an energy–aware scheduling policy should avoid throttling and, at the same time, should limit the number of migrations required to achieve this goal.

We propose two energy–aware scheduling policies that achieve both objectives. In situations with more than one task per CPU, we employ *energy balancing*. This scheduling policy balances the processors' temperatures by leveling their energy consumptions. Therefore, we combine hot tasks with cool tasks on each CPU in a way that equalizes the average of the task's energy profiles in each runqueue. Once energy consumption is balanced that way, there is no further need for task migrations. However, this is only true if all processors possess the same cooling characteristics. Actually, one processor may be located closer to some cooling component, such as a fan or an air inlet, than another one and may thus be able to dissipate more energy per time unit without overheating. To keep all processors at the same temperature, we therefore do not equalize the processors' average power, but rather the ratio between this average and the maximum power a processor is able to dissipate without overheating. Section 4.4 describes energy balancing in detail.

For processors running only one task, balancing power consumption by combining tasks with different energy characteristics is not applicable. In such situations, we apply a different policy called *hot task migration*: If a single hot task is running on a processor, we migrate the task to a cool processor at the time when the hot processor's temperature reaches the limit at which throttling would start. This way, we minimize the number of migrations. After the migration, the hot task continues to run unthrottled on the cool processor. Section 4.5 delineates hot task migration more in depth.

4.1 Affinity Scheduling

In SMP systems, each task can, in principle, run on any CPU. However, after a task has been running on a CPU for some time, it has warmed up the CPU's cache. Therefore, it is beneficial to respect *processor affinity* and to resume a task on a CPU on which it ran previously, since this reduces the need for reloading the contents of the cache. *Affinity scheduling* [27, 29] takes advantage of this, and strives to avoid running tasks on different CPUs. One approach to keep tasks local to CPUs, which has been adopted by many of todays multiprocessor operating systems, consists in introducing a local runqueue for each CPU. The scheduler divides the set of runnable tasks among the runqueues, and every CPU executes tasks from its local runqueue only.

The approach of local runqueues entails the problem of *load imbalances*, i.e., unequal runqueue lengths. The more tasks a runqueue consists of, the smaller is the share of CPU time each task gets. If there are CPUs having empty runqueues while the runqueues of other CPUs consist of multiple tasks, CPU capacity is wasted. Counteracting this problem requires *load balancing*, which means migrating tasks between runqueues. General purpose operating systems strive to provide fairness to their tasks, and keep all runqueues to the same length by moving tasks from longer runqueues to shorter ones if necessary. The problem of balancing energy consumption is very similar to the one of load balancing, since balancing power consumption also requires migrating tasks between runqueues. We take advantage of this and combine energy balancing with load balancing into one algorithm (see Section 4.4).

NUMA (non–uniform memory access) systems aggravate the affinity problem by an additional dimension. They introduce *node affinity*, which is retained if a task is kept to the same node. Respecting node affinity is beneficial to performance [10]. If a task gets migrated across the node boundary, the memory the task is referencing must either be transferred to the new node, or the task has to do inter–node accesses. Both possibilities incur performance penalties. Therefore, load balancing between CPUs belonging to the same node should be preferred over load balancing between CPUs belonging to different nodes. Similarly, in SMT systems, load balancing should preferably be done between sibling CPUs, since 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 [1].

A scheduler domain consists of two or more *CPU groups*. A CPU group is a set of CPUs. Scheduler domains are stacked in a hierarchical fashion to mirror the system's topology. The higher the level in this domain hierarchy, the costlier are the balancing operations within a domain. The domain hierarchy enables a scheduler to do *hierarchical load balancing* between the domain's groups and to resolve imbalances within the lowest domain possible.

For example, in a system consisting of two NUMA nodes with four processors each, and with each processor being two–way simultaneously multithreaded, there are three levels in the domain

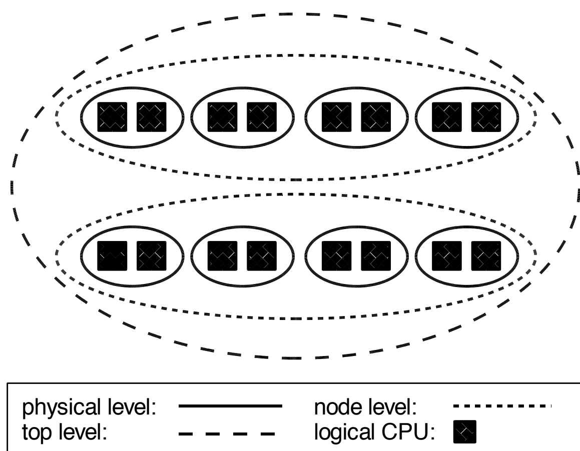

Figure 1: Example for scheduler domains

hierarchy (see Figure 1): The domains on the lowest level collect the logical CPUs residing on the same physical processor. The domains on the second level collect the CPUs belonging to the same node, and the top level domain finally contains all CPUs of the system.

Our energy–aware scheduling policies respect the domain hierarchy when doing task migrations for energy reasons, to keep the overhead required for those migrations as low as possible.

4.2 Thermal Model

Our energy–aware scheduling policies influence the processors' temperatures by directing power consumption. This requires knowledge about how temperature is related to power consumption. For this purpose, we provide the scheduler with a simple thermal model of the processor and its heat sink. We use the energy values delivered by energy estimation using event monitoring counters (see Section 3.2) as input values to this thermal model. The error resulting from estimating energy and then estimating temperature based on the energy estimate is smaller than one Kelvin for real–world applications.

Our model, which is described in detail in [8], consists of one thermal resistor and one thermal capacitor (see Figure 2). The resistor models the thermal resistance of the heat sink, which delivers heat from the processor to the ambient air. The capacitor models the thermal capacitance of the processor chip and the heat sink storing energy. This network yields an exponential function describing temperature, comparable to the exponential function describing the loading and unloading of an electrical capacitor.

The model can easily be calibrated to the thermal properties of a specific processor; we only have to adapt the coefficients of the exponential function to the real behavior of the processor. We did this by starting a task producing a maximum of heat on a processor formerly idle, recording the temperature values over time and fitting an exponential function to the experimental data. Although we did the experiments presented in Section 6 using a static setup and off–line calibration, calibration could also be done on–line by simultaneously observing temperature (read from the chip's thermal diode) and power consumption (derived from energy estimation) to account for changes in the cooling system, e.g. the activation or deactivation of additional fans, or changes in the ambient temperature.

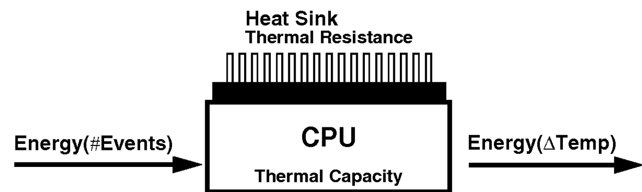

Figure 2: Thermal model

4.3 Calculation Parameters

Although processor temperature is strongly related to power consumption, the two are separate metrics with very different characteristics: Most notably, when the CPU switches between tasks with different energy profiles, power consumption reacts and changes quickly, whereas temperature only changes slowly over time.

Designing our scheduling algorithms, we noticed that algorithms based on the processors' power consumptions, since power consumption changes quickly, easily lead ping–pong effects (tasks being migrated back and forth). Scheduling algorithms only based on temperature, on the other hand, tend to over–balance, shifting all hot tasks to cool processors and all cool tasks to hot processors, so one imbalance is replaced by another imbalance in the opposite direction, requiring further migrations in the future. As a solution, the scheduling algorithms described in Sections 4.4 and 4.5 use multiple measures, describing power consumption and temperature, as input parameters:

Runqueue power

Since scheduling intervals are much shorter than the time it takes the processor and the heat sink to warm up noticeably, we take the average of the energy profiles of all tasks in a CPU's runqueue as a measure for the CPU's power consumption and call it *runqueue power*. This metric has the advantage of immediately reflecting the effect that a task migration has on the power consumption of the CPUs; migrating a task from CPU A to CPU B changes the runqueue power of CPU A as well as the runqueue power of CPU B.

Thermal power

To obtain a metric describing a processor's temperature, we use the results delivered by energy estimation as described in Section 3.2 together with our thermal model described in Section 4.2. At the end of each timeslice, when we do energy estimation for determining a task's energy profile, we re–use the knowledge about the amount of energy consumed during that timeslice to calculate a CPU–specific exponential average of energy consumption we call *thermal power*. In contrast to the task–specific energy profile, the thermal power considers any task running on a specific CPU. Since we want the thermal power to represent the processor's temperature, we calibrate it to the exponential function of our thermal model, which means choosing an appropriate weight p for the exponential average (see Equation 2) that corresponds to the time constant of the exponential function from the thermal model. This way, the course of thermal power follows temperature, but our metric still has the dimension of a power, which is important, since our scheduling algorithms (see below) need to compare a processor's thermal power to its runqueue power. Figure 3 illustrates the relationship between temperature, power, and thermal power by means of an example where power consumption rises to a higher level for some time units and then drops down to the original level again. Just like temperature, runqueue power slowly approaches

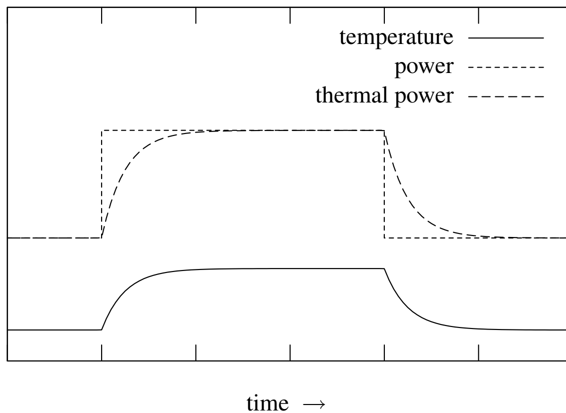

Figure 3: Relation between temperature, power, and thermal power

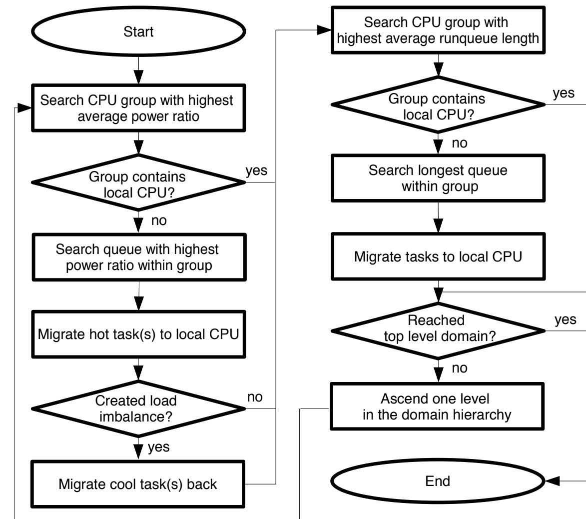

Figure 4: Energy and load balancing algorithm

the new power level, and slowly drops again after the power level has dropped.

Maximum power

We define the *maximum power* of a processor as the maximum static power consumption the processor can sustain for a long time without overheating and without having to be throttled. This definition implies that a processor whose thermal power is equal to its maximum power has reached its maximum temperature.

Runqueue power ratio

As motivated at the beginning of Section 4, we do not necessarily want to balance the processors' power consumption but rather the ratios between the power consumption and the processor–specific maximum power. We therefore define the *runqueue power ratio* as runqueue power divided by maximum power.

Thermal power ratio

Analogously, we define the *thermal power ratio* as thermal power divided by maximum power.

4.4 Energy Balancing

We apply *energy balancing* for leveling the power consumptions of CPUs whose corresponding runqueues consist of multiple tasks. 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.

As the balancing algorithm is executed on every CPU, balancing needs only be done in one direction: Linux's load balancer, e.g., only pulls in tasks from remote runqueues 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, which means migrating hot tasks from other runqueues to the local queue.

Our energy balancing algorithm uses both metrics described in the preceding section, runqueue power ratio and thermal power ratio. To avoid ping–pong effects and over–balancing, we only consider a remote queue hotter than the local queue if the remote processor has a higher temperature than the local one (represented by the thermal power ratio) and is consuming more power than the local one (represented by the runqueue power ratio). Since thermal power ratio only changes slowly, this creates a kind of hysteresis effect (it takes some time until the need to migrate a task back can arise), while runqueue power ratio, changing immediately, forbids to pull over an undue number of tasks.

Avoiding ping–pong effects also means that energy balancing decisions must be consistent with load balancing decisions 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 load balancing algorithm existing in Linux with energy balancing into one algorithm.

Since we want to resolve energy and load imbalances at the lowest cost possible, we respect the scheduler domain hierarchy. Our algorithm works in two steps, which are executed for every level in the hierarchy (see Figure 4). During the energy balancing step (left column of the figure), we strive to balance the power ratios of the domain's CPU groups by moving hot tasks from the hottest CPU group to the local CPU group. Since we do not want to create load imbalances, we migrate cool tasks back in exchange if necessary. During the load balancing step, we move tasks from the most loaded CPU group to the local CPU group. To avoid creating energy imbalances, we move hot tasks, if the remote CPU group is hotter, or cool tasks if the remote CPU group is cooler than the local group.

4.5 Hot Task Migration

We apply hot task migration to migrate a single hot task from a processor that is about to reach a critical temperature. If a CPU's thermal power comes closer to the CPU's maximum power than a predefined threshold — which, due to the calibration of these metrics to the CPU's thermal characteristics, means that the processor has nearly reached its temperature limit — and the CPU's runqueue consists of one task only, we migrate the task to a cooler

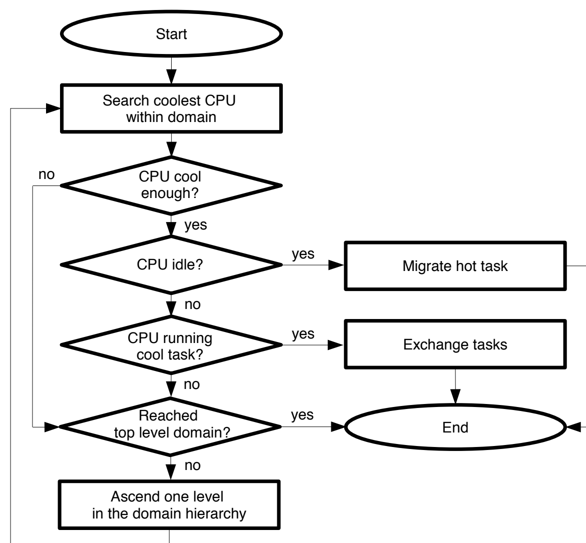

Figure 5: Hot task migration algorithm

CPU, provided there is a suitable one. The destination CPU must be considerably cooler than the source CPU to limit the frequency at which hot tasks are migrated. Hence we define a threshold by which the thermal powers of source and destination CPU must at least differ.

The destination CPU can either be an idle CPU or a CPU running a cool task. In the second case, we migrate the cool task back to the hot CPU in exchange, since we do not want to create a load imbalance. Figure 5 depicts the hot task migration algorithm. Again, to keep the cost of the migration as low as possible, the scheduler traverses the domain hierarchy, similarly to energy balancing, bottom–up to find a suitable destination CPU. 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 will in turn have to be throttled.

4.6 Initial Task Placement

In any case, whether energy balancing or hot task migration is applicable, choosing the right CPU for a newly started task is important. An inauspicious placement entails the need for migrations in the near future. For tasks running only for a short time, placing a task on the right CPU from the start is a prerequisite for energy balancing to work at all, since those tasks might terminate prior to being migrated for the first time.

As stated in Section 3.1, we cannot know the energy characteristics of a task in advance because of the different phases occurring during the task run, which depend on input data. Most tasks, however, do some initialization before processing any input data. Therefore, if a task is started, its initial behavior is independent of the data that the task processes. We take advantage of this and store the amount of energy a task consumes during its first timeslice in a hash table indexed by the inode number of the task's corresponding binary file. If a new task is started from the same binary, we initialize its energy profile from the hash table. For binaries started for the very first time, we use a default value.

Using this initial energy profile, the scheduler chooses a fitting CPU for the task. First of all, we want to avoid creating load imbalances. Therefore, a CPU is only eligible for running the new task if there is no other CPU currently running fewer tasks. For each of the CPUs in question, the scheduler calculates what the CPU's runqueue power ratio would be if the new task were assigned to that CPU. Since our goal is to have the same ratio for all CPUs, we assign the new task to the CPU whose ratio, including the new task, comes closest to the average ratio of all CPUs. This way, hot tasks get assigned to cool CPUs and cool tasks get assigned to hot CPUs.

4.7 Simultaneous Multithreading

With some adaptations, energy–aware scheduling is also applicable to systems with simultaneously multithreaded processors. If the processor's event monitoring counters are able to distinguish what logical CPU caused an event, which is the case for most events on the Pentium 4, we are able to estimate energy consumption separately for the tasks running in parallel on an SMT processor.

Since the logical CPUs of an SMT 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: We mark the scheduler domains on the lowest level, which encompass all logical CPUs belonging to the same physical processor, with a special flag. This tells the scheduler not to do energy balancing, so it skips the energy balancing step for those domains. Load balancing, on the other hand, must still be done between sibling CPUs, but the energy restrictions for load balancing mentioned in Section 4.4 do not apply.

Since energy balancing must be done between logical CPUs belonging to different physical processors, we still need the calculation parameters for energy–aware scheduling for every logical CPU. Therefore, we calculate runqueue power and thermal power for each logical CPU, and divide the maximum power a physical processor can endure without overheating between all its logical CPUs.

Note that due to energy balancing not being done between sibling CPUs, it may be that one logical CPU operates above its maximum power 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 logical CPU operating above the maximum power are not necessarily transferred to another 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 thermal powers of all logical CPUs belonging to a physical processor is greater than the allowed maximum power 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 modified a Linux 2.6.10 kernel to support energy estimation, task energy profiles, and energy–aware scheduling. Our energy estimator, which we integrated into the kernel, reads the CPU's event counters on every task switch and at the end of each timeslice, transforming the counter values into energy values as described in Section 3.2.

We also integrated an energy profiling component calculating the tasks' energy profiles and the CPU specific power ratios from the energy values delivered by the estimator. We store a task's energy profile in the `task_struct` data structure, which Linux uses to describe tasks. We also extended the `runqueue` data structure to

Table 2: Programs used for the tests

program	power	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

hold the runqueue power and thermal power, as well as the allowed maximum power.

To support energy–aware scheduling, we modified Linux's scheduler in three places: We replaced the load balancing algorithm with the combined energy–load balancing algorithm described in Section 4.4, we added a mechanism for migrating hot tasks to cool CPUs as described in Section 4.5, and we modified the policy for assigning newly started tasks to CPUs after a `fork()`/`exec()` system call as described in Section 4.6. In total, our modifications sum up to roughly 2000 lines of C–code.

6. EVALUATION

We evaluated our implementation with a set of different workloads to show how energy–aware scheduling improves the performance of a system by reducing the need for throttling. We ran our modified Linux kernel on an IBM xSeries 445 8–way Pentium 4 Xeon multiprocessor system (2.2 *GHz* each processor). The system consists of two NUMA nodes with four two–way multithreaded processors on each node.

6.1 Energy Balancing

To test energy balancing, we set the maximum power of all CPUs to $60W$. Hence, our algorithm should balance power consumption evenly across all CPUs. For the first test, we disabled simultaneous multithreading.

We ran a mixed workload consisting of six different programs (see Table 2) and started each program thrice, 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 6 depicts the thermal power of the eight processors during the test run. Because we use an exponential average for calculating this metric, the curves 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. If there is a temperature limit corresponding to a thermal power of $50W$ (denoted by the big dashed line), some of the time some CPUs operate above the limit and have to be throttled.

Figure 7 shows the thermal power of the eight processors with energy balancing enabled. Although there is a variation in the overall power consumption because of the non–static behavior of the OpenSSL benchmark, the width of the array of curves is limited. This results in all CPUs operating below the temperature limit all of the time, so we can avoid throttling.

We ran the experiments several times for 15 minutes, and counted the number of task migrations during the experiments. On average, there were 3.3 migrations with energy balancing disabled and 32 migrations with energy balancing enabled. Although this is an increase by a factor of nearly ten, the overhead of the additional migrations is small compared to the benefit of avoiding processor throttling. Considering the total of 18 running tasks we had in our experiment, 32 migrations means that on average each task was migrated less than twice during the 15 minutes, so the overhead is negligible.

We did the same experiments with SMT enabled to test the impact of simultaneous multithreading on energy balancing. Since with SMT, there are 16 logical CPUs instead of 8, we started each program six times, for a total of 36 tasks. The results are similar to those of the experiments with SMT disabled. On average, there are 9.8 migrations with energy balancing disabled and 87 migrations with energy balancing enabled.

6.2 Temperature Control

We applied our energy–aware scheduling policy in combination with temperature control by means of throttling to be able to quantify the benefits of energy–aware scheduling. We calibrated the parameters of our thermal model separately for each of the eight processors to account for their individual thermal properties. Without temperature control, the maximum processor temperature measured for our workload was 45°C. We chose an artificial limit temperature of 38°C to create a need for throttling. Whenever a CPU's

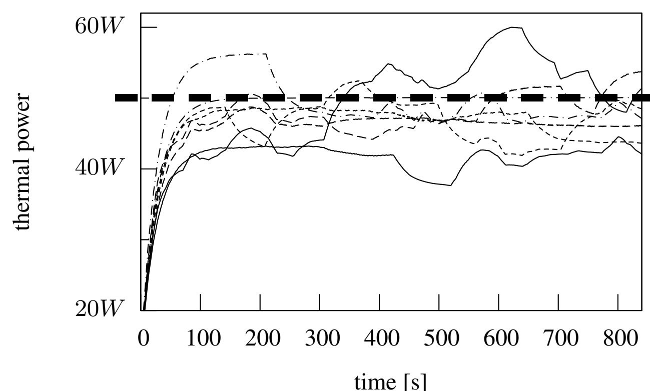

Figure 6: Thermal power of the eight CPUs with energy balancing disabled

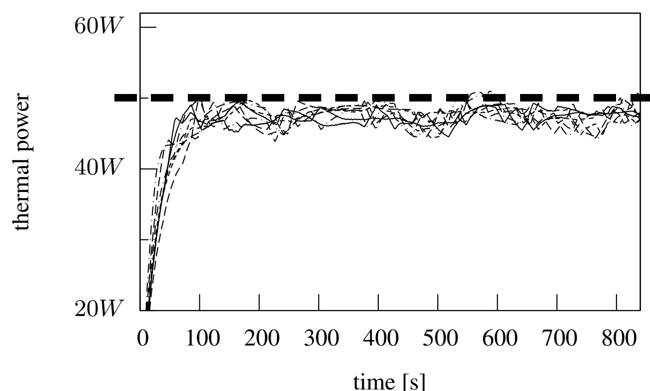

Figure 7: Thermal Power of the eight CPUs with energy balancing enabled

Table 3: CPU throttling percentage

logical 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%**

thermal power rose above the value corresponding to a temperature of 38°C, we throttled the CPU by executing the `hlt` instruction.

Again, we ran the test first with energy balancing disabled and then with energy balancing enabled. Table 3 shows the percentages of the time the CPUs were throttled during 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 properties. (Their temperature does not exceed 38°C even if the hottest task, bitcnts, is executed on them.)

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 properties. The processors with poorer thermal properties, 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) increases by 4.7% with energy–aware scheduling enabled.

We did the same experiment using a workload of short running tasks with execution times of less than a second. In this case, initial task placement is most essential. We obtained similar results; with energy–aware task placement the throughput increases by 4.9%.

6.3 Dependence on the Workload

By how much energy–aware scheduling increases throughput depends on the workload, or more precisely, on how different the energy characteristics of the tasks are. The more heterogeneous a workload is in terms of energy profiles, the more room the energy–aware scheduler has for directing the CPUs' power consumptions by assigning tasks to CPUs.

We measured how much energy–aware scheduling increases the throughput for workloads of different homogeneity (see Figure 8). We created workloads from three applications: bitcnts with a high power consumption, pushpop with a medium power consumption, and memrw with a low power consumption. We ran the tests with simultaneous multithreading disabled and used workloads consisting of 18 running tasks. We started with an heterogeneous workload consisting of nine instances of memrw and nine instances of bitcnts. For the following tests, we successively replaced one instance of memrw and one instance of bitcnts with an instance of pushpop, until we arrived at a totally homogeneous workload consisting of 18 instances of pushpop.

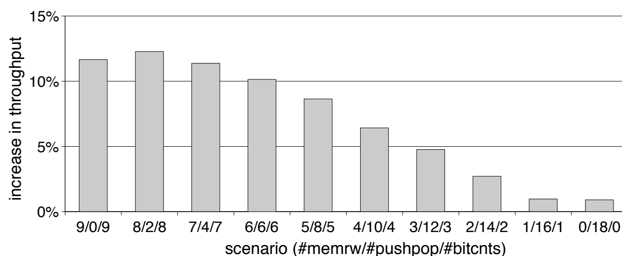

Figure 8: Dependence of throughput on the workload

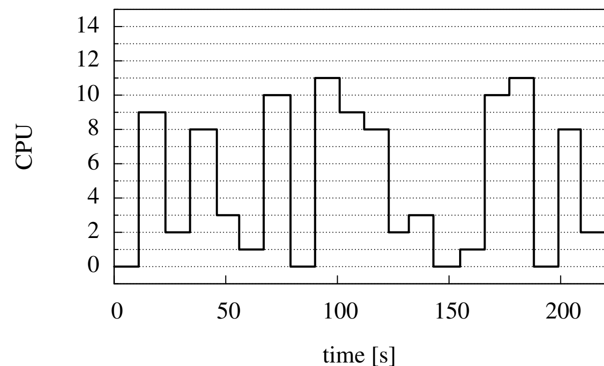

Figure 9: Hot task migration of a single task

As expected, the increase in throughput is best with heterogeneous workloads, since our energy–aware scheduler can adapt the processors' power consumptions to their thermal properties and run hot tasks on processors with better thermal properties and cool tasks on processors with worse thermal properties. Energy–aware scheduling achieves the highest increase in throughput (12.3%) for the workload consisting of eight instances of memrw and bitcnts, and two instances of pushpop. The increase for this workload is even higher than the increase for the workload consisting only of memrw and bitcnts, since some of the processors in our system possess medium thermal properties, and energy–aware scheduling can therefore best balance temperature when there are some tasks with medium power consumptions to run on these processors. With homogeneous workloads, where all tasks possess the same energy characteristics, energy–aware scheduling has almost no benefit, since the scheduler has no chance of influencing the processors' power consumptions by task migrations.

6.4 Hot Task Migration

Up to now, all the results we presented were obtained with workloads consisting of many tasks that kept all CPUs busy. In this section, we present evaluations done with workloads that leave some CPUs idle. In this case, our energy–aware scheduler applies the hot task migration policy described in Section 4.5.

For the first test, we allowed each physical processor 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 thermal powers of the two sibling CPUs (one idle, one executing bitcnts) to rise above $40W$. The bitcnts task is then migrated elsewhere by the hot task migration mechanism. Figure 9 shows for each point in time the logical CPU on which bitcnts was running.

Two things are worth noting: Firstly, bitcnts is never migrated to a sibling CPU on the same physical processor. (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. However, after bitcnts has taken one full turn, the CPU on which it executed first has cooled down enough to avoid inter–node migration.

If bitcnts were executed on one processor all of the time, as is the case without hot task migration, 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, which would be the ideal case. However, the processors in our test system consume $13.6W$ when put into a sleep state by executing the `hlt` instruction. This increases the percentage of time the processor needs to spend sleeping to enforce the $40W$ limit. Therefore, we even measured a 43% decrease in execution time with hot task migration, which corresponds to an increase in throughput of 76%. Even if we set the maximum power of the processors to $50W$, hot task migration still results in a 21% decrease of the execution time (27% increase in throughput).

For this scenario with only one task, energy–aware scheduling yields a much higher increase in throughput compared to the scenarios of the previous section: Since we always have a cool idle CPU where we can migrate the hot task, we can completely get rid of throttling. Idle processors consume considerably less power than even processors running cool tasks. Therefore, a system with some idle CPUs tends to show greater thermal imbalances that can be taken advantage of by energy–aware scheduling than a system with all CPUs busy.

When a hot processor becomes idle because a hot task has been migrated to another processor, it takes some time for the processor to cool down. Therefore, if there are multiple tasks running in the system, there might not always be a cool CPU available, so the hot task has to stay on the hot CPU, which in turn has to be throttled. Figure 10 shows the throughput of a system running multiple instances of the bitcnts program relative to the throughput of a system with energy–aware scheduling disabled. With two running tasks, energy–aware scheduling yields the same increase in throughput as with only one task. With two tasks, the first one runs on CPUs 0 to 3 in turns as in the scenario with one task, whereas the second one runs on CPUs 4 to 7. If there are more than two tasks, there are situations when there is no cool target processor because the processors do not cool down fast enough, and some of the time, tasks have to run on throttled processors. Therefore, the more tasks are running, the smaller is the increase in throughput. With eight (or more) running tasks, after some time, all physical processors are hot, and there is never a target CPU suitable for hot task migration. This results in a throughput identical to the one of a system without energy–aware scheduling.

6.5 Analysis

To assess the benefits of energy–aware scheduling, we have to 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 a difference of three orders of magnitude, 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.

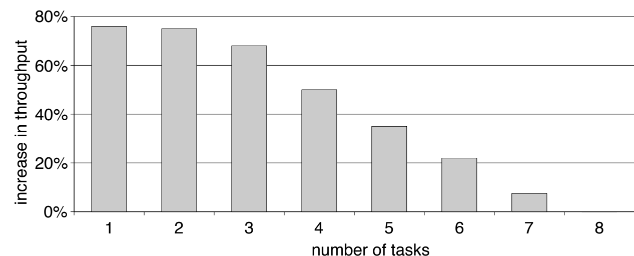

Figure 10: Hot task migration — throughput with multiple tasks

How big the benefit of energy–aware scheduling actually is depends on the diversity of the energy characteristics of the tasks that the system executes. For typical scenarios with mixed workloads, the throughput increased by about 5% when we enabled energy–aware scheduling. The corner cases are workloads extremely homogeneous in terms of power consumption, where all tasks possess the same energy characteristics, and extremely heterogeneous workloads with some tasks having a very high and other tasks having a very low power consumption. In the first case, energy–aware scheduling has no benefits, whereas in the second case, energy–aware scheduling has maximum benefits.

A special case of an extremely heterogeneous workload is a workload which does not utilize all of a system's CPUs, because the power consumption of idle CPUs is extremely low. Migrating tasks from hot CPUs to idle CPUs instead of throttling the hot CPUs increases performance substantially.

7. LIMITATIONS AND FUTURE WORK

The main limitation of energy–aware scheduling lies in the fact 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, chip temperature is likely to 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, and could characterize tasks not only by their power consumption, but also by the location at which energy is dissipated. This way, energy–aware scheduling would even be beneficial for tasks having the same power consumption, if they dissipate energy at different functional units, as is the case with floating point and integer applications.

We believe that energy–aware scheduling is also applicable to chip multiprocessors (CMP), which will probably become common in future systems. Compared to multiple one–core chips, having multiple cores on the same chip leads to greater thermal stress, since the heat is dissipated within a smaller area. Migrating tasks between individual cores on a CMP for energy reasons is beneficial because different cores on the same chip can have different tem-

peratures. Since our energy–aware scheduling framework uses the scheduler domains abstraction, extending energy–aware scheduling for use on a CMP is a matter of adding an additional layer to the domain hierarchy.

8. CONCLUSIONS

In this paper, we showed that the characterization of individual tasks by their power consumption, determined by on–line analysis using event monitoring counters, can be used to influence the power consumptions and temperatures of the processors in an SMP system via scheduling decisions. By employing an energy–aware scheduling policy, the operating system is able to reduce thermal imbalances between the system's processors and thus to mitigate the need for throttling processors. Therefore, the two main contributions of this work are:

- Task energy profiles as a means for characterizing individual tasks on–line by their power consumption
- Energy–aware scheduling as a means for balancing the processors' power consumptions and for reducing thermal imbalances.

Our evaluations show that the energy–aware scheduling policy achieves its goal and, for a lot of scenarios, results in increased throughput. Compared to the benefit of avoiding CPU throttling, the overhead incurred by hot task migration and energy balancing is negligible.

9. REFERENCES

[1] `linux/Documentation/sched-domains.txt`. Documentation shipped with the Linux source code, 2005.

[2] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*, 2001.

[3] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, 2001.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[5] 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)*, 1999.

[6] F. Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of the 9th ACM SIGOPS European Workshop*, 2000.

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

[8] 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)*, 2003.

[9] J. R. Bulpin and I. A. Pratt. Hyper-threading aware process scheduling heuristics. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[10] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[11] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[12] W. M. Felter, T. W. Keller, M. D. Kistler, C. Lefurgy, K. Rajamani, R. Rajamony, F. L. Rawson, B. A. Smith, and E. V. Hensbergen. On the performance and use of dense servers. *IBM Journal of Research and Development*, 47(5/6), 2003.

[13] K. Flautner and T. Mudge. Vertigo: automatic performance-setting for linux. *SIGOPS Oper. Syst. Rev.*, 36(SI):105–116, 2002.

[14] S. Ghiasi, T. Keller, and F. Rawson. Scheduling for heterogeneous processors in server systems. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, 2005.

[15] 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):260–270, 2004.

[16] S. Heo, K. Barr, and K. Asanovi. Reducing power density through activity migration. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISPLED'03)*, 2003.

[17] 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)*, 2004.

[18] 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*, 2003.

[19] R. Joseph and M. Martonosi. Run–time power estimation in high–performance microprocessors. In *The International Symposium on Low Power Electronics and Design (ISLPED'01)*, 2001.

[20] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single–ISA heterogeneous multi–core architectures: The potential for processor power reduction. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[21] 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)*, 2005.

[22] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12), 2003.

[23] P. Pillai and K. G. Shin. Real–time dynamic voltage scaling for low–power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[24] E. Rohou and M. D. Smith. Dynamically managing processor temperature and power. In *Proceedings of the 2nd Workshop on Feedback–Directed Optimization*, 1999.

[25] 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)*, 2003.

[26] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *SIGOPS Oper. Syst. Rev.*, 34(5):234–244, 2000.

[27] M. S. Squillante and E. D. Lazowska. Using processor–cache affinity information in shared–memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, 1993.

[28] 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*, 2002.

[29] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the performance of cache–affinity scheduling in shared–memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, 1995.

[30] 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'02)*, 2002.

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

[32] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: managing energy as a first class operating system resource. *SIGPLAN Not.*, 37(10):123–132, 2002.