Task Activity Vectors: A New Metric for Temperature-Aware Scheduling

Andreas Merkel
merkela@ira.uka.de

Frank Bellosa
bellosa@ira.uka.de

System Architecture Group
University of Karlsruhe
76128 Karlsruhe, Germany

ABSTRACT

Non-uniform utilization of functional units in combination with hardware mechanisms such as clock gating leads to different power consumptions in different parts of a processor chip. This in turn leads to non-uniform temperature distributions and problematic local hotspots, depending on the characteristics of the currently running task. The operating system's scheduler, responsible for deciding which task to run at what time, can influence temperature distribution. Our work investigates what the operating system can do to alleviate the problem of hotspots. We propose task activity vectors describing which functional units a task uses to what degree. With the knowledge provided by these vectors, the scheduler can schedule tasks using different units successively, distribute tasks using a particular unit excessively over the system's processors, or mix tasks using different units on a SMT processor. We implemented several vector-based scheduling strategies for Linux. Our evaluations show that vector-based scheduling considerably reduces hotspots.

Categories and Subject Descriptors

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

General Terms

Design, Management

Keywords

Activity Vectors, hotspot reduction, task characteristics, task migration, temperature-aware scheduling, thermal management

1. INTRODUCTION

Increasing clock rates in combination with increasing integration densities have led to an aggravation of thermal problems in recent generations of microprocessors. Strategies have been proposed to avoid overheating the processor by using features like frequency scaling, clock modulation, or halting, which reduce the switching activity of the processor and thus its power consumption [17]. All those strategies aim at keeping *CPU temperature*, which is usually understood as 'the temperature of the CPU chip', below a critical limit.

In general however, the temperature of a chip cannot be described accurately by a single temperature value. Different functional units on the chip such as arithmetic logic units (ALUs), floating point units (FPUs), or caches have different structures and thus different power densities, resulting in a non-uniform temperature distribution within the chip. Moreover, clock gating, which has been introduced to limit the overall power consumption of the chip, deactivates the clock signal for units that are currently not in use, and further increases thermal imbalances within a chip.

Since the units that are not needed for processing a certain instruction are inactive, the actual temperature distribution within a chip depends on the application running on the processor. Different programs differ in the type of instructions they execute, depending on their functionality, the way they are written, and on the compiler that was used. There are programs that issue many integer instructions, but no floating point instructions, programs that contain many branch instructions and programs that do not, programs that do many memory accesses and programs that almost only work on registers. Thus, the program that the processor currently executes influences the distribution of power density on the processor die and for that reason also the distribution of temperature. We observed variations of up to 30 Kelvin for the temperature of functional units between different applications from the SPEC CPU2006 benchmark suite on an Intel Pentium 4 Xeon processor.

Preventing thermal emergencies requires throttling the processor as soon as the temperature of the hottest part of the chip surpasses the critical temperature limit, no matter if there are other units whose temperature is far below the limit. Throttling reduces the processor's power consumption, but also its performance, and should therefore be avoided. Balancing temperature within a chip mitigates hotspots and thus also reduces the need for throttling.

As mentioned before, the temperature distribution of a chip is determined by the software that runs on the chip. For example, if the integer ALUs of a chip are near the critical temperature, and an integer task is scheduled next, the processor is likely to overheat soon, which results in throttling. On the other hand, if a floating point task is scheduled next, the ALUs remain idle and can cool down. Thus, balancing temperature within a chip, which means preferably scheduling tasks using cool units, avoids throttling.

The decisions of the operating system's scheduler influence the need for throttling a processor, which in turn influences the system's performance. In order to achieve maximum performance, the scheduler needs to know about the characteristics of each task, that is, which units of the chip a particular task uses, and to what degree it uses these units.

In this paper, we investigate to what extent the scheduler can influence the temperature distribution within a processor. We propose *task activity vectors* for characterizing tasks by their utilization of functional units on a chip. Based on these vectors, we propose and evaluate the following scheduling strategies:

- scheduling tasks that use different units successively
- distributing tasks among processors of a multiprocessor system so that each processor executes tasks that use different units
- running tasks that use different units simultaneously on multithreaded processors.

We implemented the aforementioned strategies for the Linux kernel. Using performance monitoring counters and an energy model, we estimated the power consumption of the functional units of the processor. Using the HotSpot [9] temperature simulator, we calculated the course of the temperature during our test runs to verify the benefits of our strategies.

The rest of our paper is structured as follows: Section 2 delineates the background of our work. Section 3 describes task activity vectors and presents the design of our vector-based scheduling policies. Section 4 outlines the implementation for Linux. Section 5 evaluates our design. Section 6 addresses the limitations of our approach and suggests directions for future work. Section 7 discusses related work. Finally, Section 8 concludes.

2. BACKGROUND

The scheduling policies we propose are motivated by the thermal characteristics that a processor chip exhibits. In the first part of this section, we want to elucidate these characteristics. The second part of the section describes the methodology we use for obtaining information about the temperature distribution on the chip.

2.1 Thermal Behavior of a Processor Chip

The functional units a microprocessor consists of (ALUs, FPUs, caches, register files, and so on) are usually laid out as blocks, i.e. (mostly) rectangular areas on the die. Figure 1 shows the floorplan of an Intel Pentium 4 Northwood (derived from a die photo) as an example.

From a physical point of view, the functional units are located on a silicon die, which in today's processor generations is of rather small dimensions, typically about 1cm in width and length and less than 1mm in height. The die is covered by a heat spreader consisting of thermally well conducting material and by a heat sink, which is comparably large in relation to the die and consists of copper or aluminum. In relation to its size, the die dissipates large amounts of power, which can surpass 100W in recent processors. This results in a power density of 100W/cm^2 and more, which is ten times the power density of a hot-plate.

The temperature of an object, be it a functional unit, the die as a whole, or the heat sink, is determined by the amount of energy that the object contains (referred to as *internal* or *intrinsic energy*) and its *heat capacity*. The heat capacity is a material specific constant describing how many Joules it requires to heat the object by one Kelvin.

The heat capacity of the die is small compared to its power dissipation, therefore the die (and thus the individual functional units) can heat up rather quickly. A small example shall demonstrate this:

A silicon die of 11mm$\times$12mm$\times$0.7mm has a heat capacity of 0.15J/K. If it dissipates 70W and no heat is removed from the die,

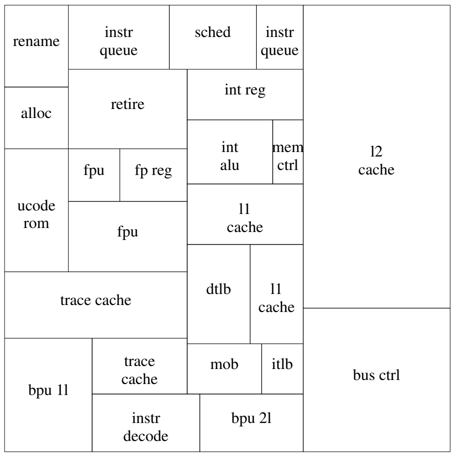

Figure 1: Floorplan of the Pentium 4 Northwood Processor

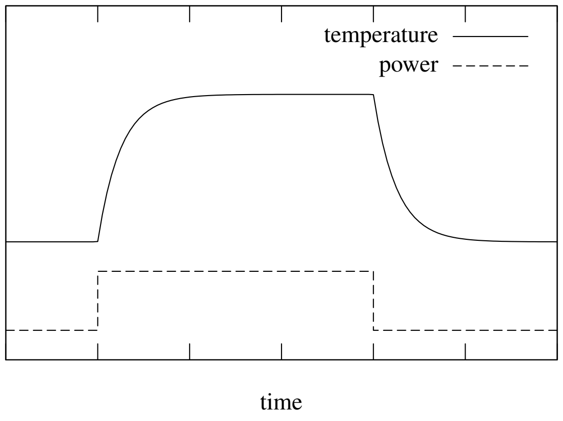

Figure 2: Dependence of Temperature from Power

it takes only 2ms for the die to heat up by 1K. In contrast to that, the heat sink has a much larger heat capacity, since it is considerably more massy, and typically takes seconds to heat by 1K.

The intrinsic energy and thus the temperature of the functional units on the die and of the heat sink can be described by similar differential equations whose solution is an exponential function [14]. As a consequence, when there is a change in power consumption, the temperature changes exponentially over time (see Figure 2). Yet, for the functional units, this change happens much more quickly than for the heat sink.

2.2 Determining Chip Temperature

Traditionally, mechanisms and strategies for avoiding overheating a processor are based on a single *chip temperature* value [3, 6, 14]. This value usually stems from a thermal diode located somewhere on the processor [10], or is assembled out of several temperature readings, if multiple sensors are present on the chip [16].

But physical sensors have several limitations: Firstly, their number is limited, since increasing the number of sensors is expensive in terms of die area. Hence not all possible hotspots can be covered. Secondly, thermal sensors often cannot be placed directly at the hottest functional units for topological reasons. Thirdly, reading temperature takes a long time and cannot be performed more than several times per second [7, 6].

As motivated above, the temperature of a functional unit can rise very quickly. The consequence of the limitations of physical sensors is that either some parts of the chip can possibly overheat without the sensors detecting it, or that the critical temperature at which throttling starts must needs be set to a lower value to compensate for the sensors reporting values that are lower than the maximum temperature somewhere on the chip. The latter, however, may lead to throttling without need and thus needlessly sacrificing performance.

Skadron et al. [13] proposes using performance monitoring counters in combination with an energy model and a thermal model to estimate the temperature of individual parts of the chip. Performance monitoring counters, special registers introduced mainly for profiling and optimization, count the occurrence of various events in the chip. The information obtained from these counters allows us to estimate the degree of utilization of individual chip units [11]. The power consumption of each unit, in turn, depends on the unit's utilization, and can thus be estimated when utilization is known. Using a thermal model, temperature estimation for the functional blocks of the chip based on their power consumption is possible [13, 7]. Estimation of power consumption from performance counters yields an estimation error of less than 10% [1, 11]; the temperature simulator HotSpot has been verified against a test chip and yielded a worst case error of 7% for temperature estimation [9].

Hence, using estimation and simulation allows us to obtain realistic temperature values with finer granularity than using hardware sensors, not only in terms of space (obtaining temperatures for each unit is possible), but also in terms of time. The latter is determined by the rate at which the performance monitoring counters are sampled, which can be done at every timer interrupt, that is up to 1000 times a second in today's systems. The methodology just described enables us to verify the benefits of our proposed scheduling policies, since it allows us to observe the course of temperature for each functional unit at a high enough temporal resolution, which is not possible using the sensors provided by the hardware.

3. VECTOR-BASED SCHEDULING

As mentioned in the introduction, the scheduling decisions of the operating system—determining what task the processor shall execute next—also determine the temperature distribution on the chip. Up to now, scheduling strategies found in general purpose operating systems like Linux or Windows are oblivious to this fact. Today's schedulers make their decisions according to criteria like task priorities, fairness or good interactive performance, but neglect the impact that the order in which tasks are executed has on temperature distribution.

At the same time, in many scheduling strategies the exact order in which tasks are executed is unspecified. For example, in round-robin scheduling, it is not relevant in which order the tasks are scheduled, as long as each task gets its timeslice in turn and all tasks make progress. The same holds true for proportional share scheduling. Even with priority-based scheduling, the order in which tasks having the same priority are scheduled is not specified.

This can be used to influence the temperature distribution of the chip without breaking the properties and objectives of the respective scheduling policy. In the remainder of this section, we will concentrate on the widely-used round-robin policy.

3.1 Task Activity Vectors

Enabling the scheduler to influence temperature in a sensible way requires providing it with information about the characteristics of the tasks it manages, i.e., what task utilizes which parts of the processor.

For this purpose, we propose *task activity vectors*. An activity vector is part of the task's runtime context. The dimension of this vector is equal to the number of functional units on the processor. Each component of the vector denotes the degree of utilization of a corresponding functional unit when the task is executed. We normalize each component to the maximum utilization the respective functional unit can exhibit. Thus, the values of the vector's components range between 0 (no utilization) and 1 (maximum utilization).

Task activity vectors make the resources the task uses on the CPU chip part of the task's runtime context, so the operating system has detailed information about the characteristics of each task.

Determining a task's activity vector requires determining the utilization of each functional unit. Information about the units' utilization could be provided by the hardware, for example via special registers. Unfortunately, this is not the case in today's processors. Therefore, we resort to the methodology proposed by Isci and Martonosi [11], and determine the degree of utilization for each functional unit using performance monitoring counters. The method suggested by Isci and Martonosi attributes events or combination of events to each functional unit and determines the utilization of the units by counting how many events occur in a given timeframe.

The activity vector of a task is not constant, but can change over time, as the task passes through different phases, e.g., runs different algorithms successively. Therefore, the operating system has to recalculate the activity vector of a task continuously. On every timer tick and on every task switch, we determine the utilization of the functional units by reading performance monitoring counters and update the activity vector of the currently running task. For each component, we use an exponentially moving average to smooth out changes in the task's behavior that are only of short duration. This way, we avoid erroneously changing a task's activity vector if the task's characteristics show temporary fluctuations.

3.2 Runqueue Sorting

We propose *runqueue sorting* as a scheduling policy that makes use of task activity vectors. The aim is to avoid the formation of hotspots. We accomplish this by choosing a task to run next that does not utilize the parts of the processor that are currently hot, provided that such a task is available.

Our policy is based on the assumption that there are CPU-local runqueues, i.e., there is a list of tasks assigned to each CPU, and that the tasks in this queue are scheduled in round-robin fashion, that is, executed for a certain period of time (a *timeslice*) and re-appended to the end of the queue thereafter. This holds true in the majority of today's operating systems, although the scheme is often slightly altered or enhanced. For instance, round-robin scheduling is often combined with a priority scheme or the runqueue is split into several sub-queues as is the case with Linux.

We propose two forms of runqueue sorting, a simple form and an enhanced form that uses a thermal model. The simple form arranges the tasks in the runqueue in a way that two successive tasks use resources of the CPU that are as complementary as possible, so the resources the previous task has used can cool down during the execution of the following task. The enhanced form considers the estimated temperatures of the CPU's functional units for choosing the next task rather than utilization.

Sorting efficiently

The set of tasks that compose a CPU's runqueue is not fixed. Tasks can terminate or block and thus be removed from the queue, and new tasks may be started. Also, the load balancer can move task between the runqueues of different CPUs. Thus, keeping the runqueues sorted causes a lot of overhead.

The alternative to sorting is searching the runqueue for a suitable task every time a scheduling decision is made. But searching is not scalable, since the time required to search for a suitable task grows with the length of the runqueue. Since scheduling a new task is a frequently invoked operation, scheduling algorithms have to be efficient. For example, Linux's scheduler is designed to satisfy an O(1) property, meaning that the cost of scheduling is independent of the number of tasks in the system [12].

We use a combination of searching and sorting to keep the overhead low. We adopt the scheme used in Linux of having two runqueues per processor, the first queue consisting of tasks waiting to be scheduled, and the second queue consisting of tasks that have been executed lately. When choosing the next task to be scheduled, we look at the first c tasks in the first queue and choose the best suited task out of them. After executing for one timeslice, we append the task to the second queue. This way, the second queue is automatically sorted in a way that tasks using complementary resources follow each other. When the first queue is empty, we switch the queues. Since we operate on a queue now that was previously sorted, there is a high probability that there is a suitable task among the first c tasks in the queue.

Additionally, having two queues ensures that all tasks are making progress: A task that has been scheduled and is appended to the second queue is not scheduled again until all other tasks still residing in the first queue have been scheduled.

Example:

We consider a scenario with two groups of tasks with similar characteristics. Tasks of type A use different resources than tasks of type B. We assume there are five tasks of type A and three tasks of type B, we have chosen $c = 3$ and the initial ordering of the runqueue is *BBBAAAAA*. Figure 3 shows how the runqueues evolve. We underline the tasks the scheduler considers for making its decision and mark the task the scheduler picks with a dot. Note that after sorting, there is still a succession of three tasks of type A since there are more tasks of this type than of type B. One might argue that it would be better to sort the queue in a fashion that distributes the tasks of type A more evenly, like *ABAABAAB*, but this is not feasible without global knowledge of the whole queue. Additionally, the functional units typically have already reached their peak temperature after one timeslice, so it does not matter whether we execute the supernumerous tasks of type A in groups of two or three.

queue 1	queue 2
$BBBAA\underline{AA\dot{A}}$	
$BBBA\underline{AA\dot{A}}$	A
$BBB\underline{AA\dot{A}}$	AA
$BB\underline{\dot{B}AA}$	AAA
$B\underline{BA\dot{A}}$	$BAAA$
$\underline{B\dot{B}A}$	$ABAAA$
$\underline{B\dot{A}}$	$BABAAA$
$\underline{\dot{B}}$	$ABABAAA$
	$BABABAAA$
$BABAB\underline{AA\dot{A}}$	
…	

Figure 3: Example for runqueue sorting

Simple sorting

In the simple form of runqueue sorting, we use only the activity vectors for deciding which tasks out of the first c tasks in the queue we pick. If the timeslices are long enough (10ms to 100ms), which is the case in most operating systems, and a task switch occurs because of timeslice expiration, the unit utilization of the previously running task is a good indicator for the units' temperatures: Because of the comparatively small thermal capacitance of the functional units, the units the task has utilized are hot and the units the task has not utilized are cold. Therefore, we choose a task that uses different units than the one that ran before.

Whether a task uses different units than another task can be inferred from the angle the tasks' activity vectors form, as we want to explicate in the following. We denote the angle between two vectors $\vec{x}$ and $\vec{y}$ by $\angle(\vec{x},\vec{y})$.

Let $\vec{a}$ be the activity vector of the task that has run on the CPU up to now and $\vec{b_i}$ with $i \in \{1,..,c\}$ the activity vectors of the tasks we consider when choosing the next task to run. We want to choose the task whose unit utilization is most different from the unit utilization of the previously running task. In the ideal case, when two tasks use completely distinct units, their activity vectors are orthogonal, so $\angle(\vec{a},\vec{b_i}) = 90°$. Since the components of all activity vectors are positive or zero, no two vectors can form an angle greater than $90°$. Therefore, we choose the task whose activity vector $\vec{b_j}$ with $j \in \{1,..,c\}$ forms the biggest angle with $\vec{a}$:

$$j = \arg\max_{i=1}^{c} \angle(\vec{a},\vec{b_i}) \tag{1}$$

With $\angle(\vec{x},\vec{y}) = \arccos \frac{\vec{x}\cdot\vec{y}}{|\vec{x}||\vec{y}|}$ we transform this to:

$$j = \arg\max_{i=1}^{c} \arccos \frac{\vec{a}\cdot\vec{b_i}}{|\vec{a}||\vec{b_i}|} \tag{2}$$

Since for $0° \leq \alpha \leq 90°$, $\arccos\ \alpha$ is strictly falling, this is equivalent to:

$$j = \arg\min_{i=1}^{c} \frac{\vec{a}\cdot\vec{b_i}}{|\vec{a}||\vec{b_i}|} \tag{3}$$

$$j = \arg\min_{i=1}^{c} \frac{\vec{a}\cdot\vec{b_i}}{|\vec{b_i}|} \tag{4}$$

Determining the length of the vectors $\vec{b_i}$ involves calculating square roots, which is expensive. Since all components of our vectors are positive, we replace $|\vec{b_i}| = \sqrt{\sum_{k=1}^{n} b_{i,k}^2}$ by $\sum_{k=1}^{n} b_{i,k}$, where $b_{i,k}$ is the k-th component of vector $\vec{b_i}$.

Hence, we choose the task with activity vector $\vec{b_j}$ to schedule next, if:

$$j = \arg\min_{i=1}^{c} \frac{\vec{a}\cdot\vec{b_i}}{\sum_{k=1}^{n} b_{i,k}} \tag{5}$$

Equation 5 does not yield exactly the same results as Equation 4. Both equations lead to choosing a task that uses, as far as possible, different units than its predecessor, which is accomplished by the scalar product in the numerator of the fraction. Omitting the square root in the denominator leads to preferring activity vectors having high values in some components and low values in other components to vectors with medium values in all components. This is advantageous for our strategy, since it 'saves' tasks with medium unit utilization for situations in which no task with complementary utilization (compared to the task just executed) can be found among the first c tasks in the queue.

Enhanced sorting

The enhanced form of runqueue sorting considers the actual temperatures of the functional units. As described in Section 2.2, these temperatures cannot be obtained directly from hardware. Therefore, we use a thermal model of the processor for estimating temperatures. Using estimated temperatures for runqueue sorting is computationally more expensive, but has the advantage of considering the physical properties of the chip, for example lateral heat spreading: Even a unit that the previously running task has not utilized can have an increased temperature if it is located next to a unit that the task has utilized heavily.

Using a thermal model also has the advantage of implicitly considering the actual period of time that a task spends on the CPU. Simple sorting assumes that tasks are exhausting their timeslice. If a task uses only a small fraction of its timeslice and then blocks or releases the CPU, the characteristics of this task are not crucial for the temperature distribution of the CPU. Enhanced sorting can cope with this, since the thermal model considers the amount of energy each tasks dissipates in each unit, and this amount is small if a task runs only for a short time.

For modeling the chip's thermal properties, we use a *compact model* as proposed by Skadron et al. [9]. Since we want to obtain temperature estimations at runtime, we choose a rather simple model of the chip, describing each functional unit as a thermal capacitor (with capacitance depending on the area the block occupies) that is linked by thermal resistors to its neighbor units. We also model the heat sink as a thermal capacitor that is linked to each unit via a thermal resistor.

Based on the utilization of the functional units, we estimate the power consumption of each unit and use it as input to the thermal model. The model delivers temperature estimations for each functional unit and is comparable to Temptor [7], another adoption of Skadron's methodology to runtime temperature estimation. Our model is simpler than Temptor and thus less exact, but requires fewer calculations.

The basic idea of enhanced runqueue sorting is to choose a task to be scheduled next that uses units having a low temperature and does not use units having a high temperature. For deciding whether a unit's temperature is currently high or low, we compare the current temperature of the unit to its average temperature. For this purpose, we calculate an exponentially moving average over the temperature of each unit (delivered by our thermal model).

Let $\vec{m}$ be a vector containing the average temperatures of all units and $\vec{t}$ a vector containing the current temperatures of the units. The difference $\vec{t} - \vec{m}$ is a vector with positive components for units that are currently hot and negative components for units that are currently cold.

We choose the task with activity vector $\vec{b}_j$ to be scheduled next, if:

$$j = \arg\min_{i=1}^{c} (\vec{t} - \vec{m}) \cdot \vec{b}_i \qquad (6)$$

If a unit is currently hotter than its average temperature, scheduling a task that uses this unit is discouraged: The corresponding component of the temperature difference $(\vec{t} - \vec{m})$ is positive; multiplication with a nonzero utilization from vector $\vec{b}_i$ yields a positive contribution to the scalar product. On the other hand, if a unit is currently colder than its average temperature, scheduling a task that uses this unit is encouraged: The corresponding component of the temperature difference is negative; multiplication with a nonzero utilization from vector $\vec{b}_i$ yields a negative contribution to the scalar product.

3.3 Activity Balancing

Runqueue sorting is effective only if a runqueue contains tasks with different characteristics. In a multiprocessor system, the scheduler can influence the contents of the runqueues by migrating tasks between CPUs. In today's operating systems, this is done for the purpose of *load balancing*, i.e., equalizing runqueue lengths by moving tasks from long runqueues to shorter ones. We use task migration to create the prerequisites for runqueue sorting. For example, if one runqueue consists solely of integer tasks and another runqueue consists solely of floating point tasks, we migrate some integer tasks to the second queue and some floating point tasks to the first queue.

Activity balancing is beneficial when done between processors that are physically different chips, as well as between different cores on one multicore chip, since it makes sure that for each location (chip or core), there are tasks having different characteristics available. For logical processors of a simultaneously multithreaded processor, however, a different strategy (described in Section 3.4) makes sense, since several logical processors dissipate heat at the same physical location.

In a previous work, we proposed *energy balancing* for equalizing the power consumptions of CPUs by migrating tasks depending on the amount of energy they consume [14]. We modify the energy balancing algorithm to do *activity balancing*, i.e., equalizing the utilization of functional units between the CPUs.

Therefore we introduce a measure called *thermal stress*, which describes whether the tasks contained in a runqueue use mainly the same units or not. The goal of activity balancing is to keep the thermal stress of all runqueues as low as possible by migrating tasks between queues.

We define the thermal stress of a CPU as the sum of the average utilization of all units whose utilization is greater than a constant l. If the average utilization of a unit is greater than l, this means that too many tasks are using the unit too heavily, and runqueue sorting cannot always find a task that does not use the unit.

Let n be the number of functional units, p the number of tasks in a runqueue, and $\vec{b}_i, i \in \{1, \ldots, p\}$ the activity vectors of the tasks in the queue. Let $\vec{m}$ be the average of the activity vectors $\vec{b}_i$:

$$\vec{m} = \frac{1}{p} \sum_{i=1}^{p} \vec{b}_i \qquad (7)$$

We define the thermal stress s of the queue formally as:

$$s = \sum_{j \in \{1,\ldots,n\} : m_j > l} m_j \qquad (8)$$

This way, units that are used by many tasks, but only to a low degree, or used only by few tasks, do not contribute to thermal stress. Hence, migration of a task that utilizes a certain unit heavily to a runqueue whose tasks do not utilize the unit does not increase the target runqueue's thermal stress.

Activity balancing works by migrating a task from one runqueue to another if this either decreases the thermal stress of both queues, or decreases the stress of one queue, while the stress of the other queue stays constant. At the same time, we ensure that activity balancing does not cause load imbalances; the approach we take in this respect is the same as in our work on energy balancing [14].

Since units with average utilization lower than l do not contribute to thermal stress, our policy encourages migrating tasks that utilize a certain unit to runqueues consisting of tasks that do not utilize the respective unit.

For each unit that is used heavily by a task, runqueue sorting requires the availability of another task that does not utilize the unit, since this permits to schedule tasks using and not using the units alternatingly. In theory, $l = \frac{1}{2}$ seems a sensible choice, since if the average utilization of a unit is $\frac{1}{2}$, this means that there are as many tasks that use the unit as there are tasks that do not. However, since tasks can use more than one functional unit, activity balancing is a multidimensional optimization problem, and $l = \frac{1}{2}$ turned out to prevent activity balancing in many situations when no configuration can be found that keeps unit utilization below $\frac{1}{2}$.

Empirically, we found $l = \frac{2}{3}$ to be a good choice. Lower values are too restrictive and prevent activity balancing too often, whereas higher values lead to runqueues containing too many tasks with similar characteristics.

3.4 Activity Unbalancing

Multithreaded CPUs offer multiple logical processors that make use of the functional units of the same physical chip. Logical processors belonging to the same physical processor are termed *siblings*.

Gomaa et al. [5] have shown that regarding thermal constraints, multithreaded chips can be used most efficiently if siblings execute tasks that use different units (and hence possess different activity vectors). With our approach, we want to verify that the principle of running tasks with different characteristics together can be achieved in a real operating system by using automatic task characterization (activity vectors) and a suitable scheduling policy (activity unbalancing).

From the operating system's point of view, the logical processors of a physical chip behave like individual processors, and most operating systems treat them this way in many respects. In particular, scheduling decisions happen independently for each logical processor, and operating systems providing CPU-local runqueues typically provide a separate runqueue for each logical processor.

Scheduling decisions for different logical processors do not necessarily happen at the same time: Tasks on different logical processors can block or release the CPU at different points in time. Therefore, it is difficult to synchronize scheduling decisions and to enforce that siblings do not execute tasks with similar activity vectors simultaneously.

We solve this problem by using task migrations to arrange the runqueues of siblings in a way that the tasks in a runqueue all use different units than the tasks in the other runqueues, if this is possible. For example, on a two-way multithreaded processor, one runqueue might contain only cache intensive tasks and memory intensive tasks, and the other runqueue only floating point tasks and integer tasks. This way, it does not matter if scheduling decisions happen independently for the siblings, since no matter which task is chosen, it is guaranteed to use different units than the task running on the sibling.

A policy different from activity balancing must be applied for sibling processors, since our aim is the opposite: Rather than making all runqueues consist of tasks with different characteristics, we want to divide all available tasks into subsets that use mutually distinct units. Therefore, we propose a strategy for *activity unbalancing*.

The strategy is based on a measure we call *diversity*. The diversity of two runqueues q_1 and q_2 denotes whether the tasks in q_1 mainly use the same units as the tasks in q_2 or not.

Let n be the number of functional units on the chip and $\vec{m}$ the average of the activity vectors of the tasks from q_1, as defined by Equation 7, and $\vec{o}$ the average of the activity vectors of the tasks from q_2, respectively. The diversity $d(q_1, q_2)$ is then defined by the following equation:

$$d(q_1, q_2) = \sum_{j=1}^{n} |m_j - o_j| \tag{9}$$

Hence, if two siblings use a functional unit to a different degree, i.e., most tasks on one sibling utilize the unit heavily, while most tasks on the other hardly use it, this causes a high contribution to the diversity of the siblings.

Activity unbalancing works by migrating a task from one sibling to another, if this increases their diversity. In systems consisting of multiple multithreaded chips, we perform activity unbalancing between runqueues belonging to siblings, but activity balancing between runqueues belonging to different physical processors, since this ensures that there are tasks with different characteristics available on each physical processor to distribute between the logical processors.

Besides running tasks with different characteristics simultaneously on siblings, we additionally perform runqueue sorting on each sibling. This is beneficial, since typically, the number of functional units on a processor is greater than the number of logical processors. Hence, even if we distribute the tasks to the runqueues in a way that the logical processors always use mutually different units, the units that the tasks in a particular runqueue are using may still differ from task to task, so that runqueue sorting is still beneficial.

4. LINUX IMPLEMENTATION

We extended a Linux 2.6.16 kernel to support scheduling based on activity vectors. We modified the kernel in the following ways:

- We enhanced the `task_struct` data structure that Linux uses to describe a task by fields describing the task's activity vector.
- We implemented the mechanism described by Isci and Martonosi [11] to determine unit utilization (and power consumption) by evaluating performance monitoring counters. We enhanced the mechanism for supporting multithreaded processors, which involves multiplexing counters, since there is only one set of performance monitoring counters per physical processor.
- We implemented temperature estimation using a compact model to obtain temperatures for each functional unit, which is required for the extended version of runqueue sorting.
- We implemented the scheduling strategies described in Section 3. In the following two subsections, we delineate how we modified Linux's scheduler to implement our policies.

4.1 Runqueue Sorting

Linux pursues a scheduling policy combining round-robin-scheduling with priorities. The scheduler assigns longer timeslices to tasks with higher priorities and schedules tasks with higher priorities before tasks with lower priorities. However, the scheduler avoids starvation of low-priority tasks by making sure that no high-priority task executes for a second timeslice unless all lower-priority tasks have been scheduled in the time in between.

This is achieved by introducing two data structures for storing ready-to-run tasks, one containing tasks that have already executed (the *expired array*) and one for tasks that have not executed yet (the *active array*). Both arrays consist of linked lists of tasks, one for each priority. When making a scheduling decision, the scheduler chooses the first task of the linked list with the highest priority from the active array. After executing, the task is enqueued in the linked list of the expired array that corresponds to its priority. When the list with the highest priority is empty, the scheduler resorts to the second highest priority and so on. When all lists of the active array are empty, the scheduler exchanges expired and active array.

To support runqueue sorting, we modified the policy for choosing the next task to schedule. Instead of choosing the first task from the queue with the highest priority in the active array, we look at the first c tasks in the queue with the highest priority. If the queue with the highest priority contains less than c tasks, we also consider tasks from the queue with the second highest priority and so forth. We choose the task that suits best according to our metric (Equation 5 or 6), execute it for one timeslice, and append it to the expired array. In our implementation, we chose $c = 4$.

Considering tasks from lower priority queues softens the priority scheme of Linux, since tasks having lower priorities may be scheduled prior to tasks with higher priorities. However, it increases the chances that a suitable task is found, if there are only few tasks of a given priority. Additionally, priorities in Linux are already designed to be soft priorities only. Native Linux also schedules a lower priority task although a higher priority task is in the ready state, if the low priority task is it the active array and the high priority task is in the expired array.

4.2 Activity Balancing/Unbalancing

For implementing activity balancing between physical processors and activity unbalancing between siblings, we modified Linux's load balancer to consider the metrics described by Equations 8 and 9 in addition to load.

We took advantage of the *scheduler domain* hierarchy that Linux uses to represent the topology of the system. The hierarchy defines groups of CPUs at different levels, e.g. groups comprising all siblings residing on the same physical processor, groups comprising all CPUs residing on the same NUMA node, and so on. Within groups consisting of siblings, we perform activity unbalancing, and migrate tasks if this increases diversity, but between CPUs not belonging to the same group of siblings, we perform activity balancing, i.e., migrate tasks if this decreases thermal stress.

We also consider the aforementioned metrics when choosing a CPU for starting a new task after a `fork()`/`execve()` system call.

5. EVALUATION

5.1 Setup and Methodology

We evaluated our implementation with a set of different workloads composed of programs taken from the SPEC CPU 2006 suite, a benchmark suite consisting of several compute intensive integer and floating point benchmarks derived from real user applications.

As described in Section 2.2, hardware sensors are not suitable for verifying the impact of our approach, namely the reduction of hotspots. Temperature measurement via external sensors such as laser thermometers is not applicable for similar reasons, e.g., it is not possible to operate the chip without a heat sink attached and hence the chip is not accessible for temperature measurements.

For these reasons, we use a combination of real hardware and simulation: We let our modified Linux kernel run on real physical hardware. We use an IBM xSeries 445 eight-way multiprocessor system (eight Pentium 4 Xeon Gallatin processors with 2.2 GHz each processor). The system consists of two NUMA nodes with four two-way multithreaded processors on each node.

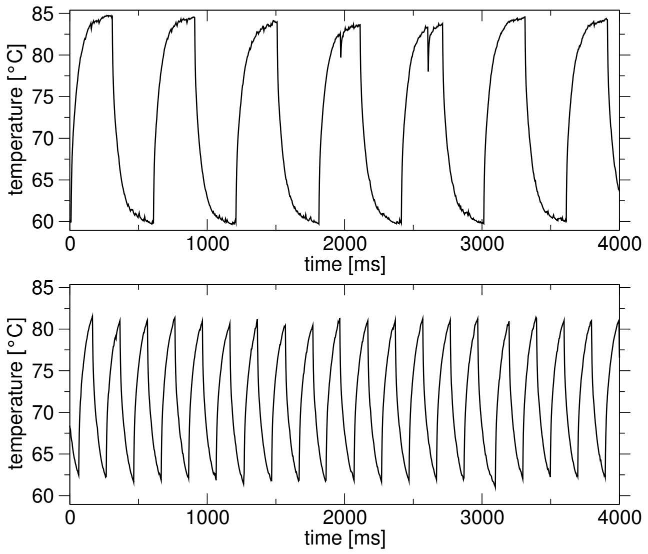

Figure 4: Temperature of floating point registers with (bottom) and without (top) runqueue sorting

For investigating temperature, we resort to simulation using the HotSpot [9] temperature simulator. During the test runs, we sample the utilization of each functional unit of the CPU. From unit utilization, we estimate the power consumption of each unit. Afterwards, we use the power estimates as input to the HotSpot simulator.

Since we had no floorplan for the Pentium 4 Xeon Gallatin processor at our disposal, we estimated temperature based on the floorplan of a Pentium 4 Northwood processor, which has the same feature size and also the same amount of L1, L2 and trace cache, but lacks the L3 cache of the Gallatin processor. Since the power consumption of the L3 cache is moderate in relation to the area the cache occupies, the L3 cache is no hotspot and can therefore legitimately be neglected.

We sampled unit utilization at millisecond granularity. To limit the amount of sampled data and to shorten the time required for the temperature simulations, we took samples only every 100th second. Thus, during a test run, the programs ran 99 seconds without sampling, then, during the following second, we took 1000 samples, let the program run for 99 seconds without sampling again, and so forth.

5.2 Runqueue Sorting

To demonstrate the benefits of runqueue sorting, we selected two SPEC benchmarks that use complementary resources. `hmmer` uses mainly the integer units (integer ALUs and integer register file) and the L1 cache. When `hmmer` is running alone, the hottest unit on the chip is the integer register file. `namd` mainly uses the floating point units; when `namd` is running, the hottest unit is the floating point register file.

We started three instances of `hmmer` and three instances of `namd` simultaneously. Since we wanted the tasks to run on one CPU together, we disabled all CPUs of the system but one.

Figure 4 shows the effect of runqueue sorting. The figure depicts the course of temperature over time for the floating point register file, which reached the highest temperature of all units during the test.

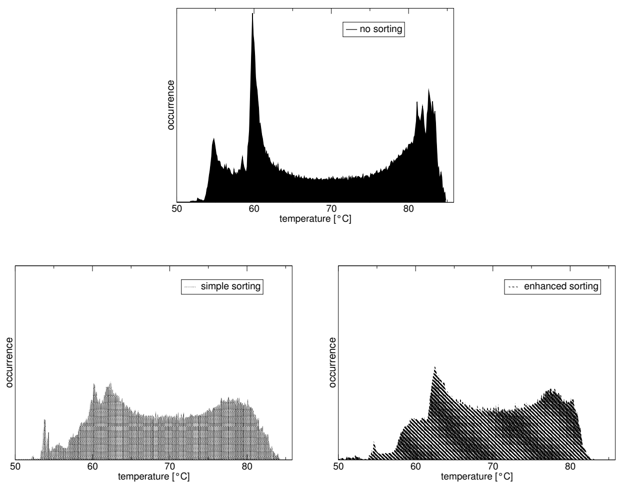

Figure 5: Temperature of floating point registers (hmmer + namd)

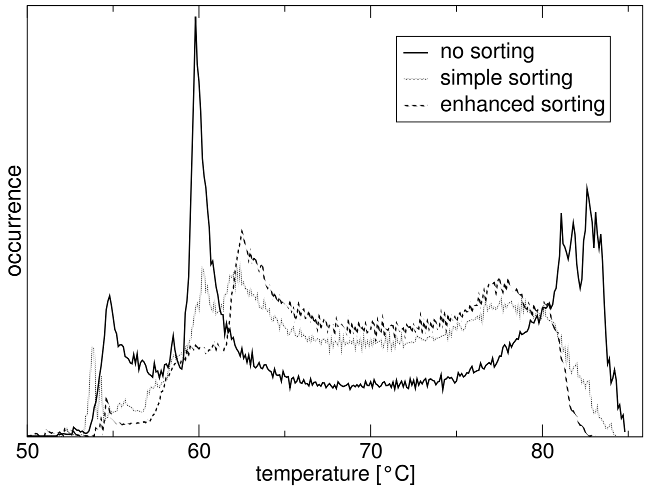

Figure 6: Temperature of floating point registers (hmmer + namd, combination of the histograms from Figure 5)

Without runqueue sorting, the order in which the tasks are scheduled is arbitrary. In the most unfavorable situation, which is shown it the top half of the figure, three timeslices in a row are assigned to the three instances of `namd`, followed by three timeslices assigned to `hmmer`. Whenever an instance of `namd` is running, the temperature increases, whereas whenever `hmmer` is running, the temperature decreases. Since a timeslice is 100ms long, temperature rises/decreases for 300ms, respectively.

With runqueue sorting enabled, the scheduler arranges the tasks in a way that whenever an instance of `namd` has been executed for one timeslice, an instance of `hmmer` gets scheduled during the next timeslice. The effect of this can be seen in the bottom half of the figure: The time during which the floating point register file's temperature increases is one timeslice at most, since after this timeslice, the scheduler picks a task that does not utilize the floating point units. Therefore, the temperature of the register file does not rise as high as it does without runqueue sorting. The effect on the other units' temperatures is similar.

The benefits of runqueue sorting become apparent in a histogram displaying the frequency of occurrence for the temperature values observed during the test run. Figure 5 shows histograms of the temperature of the floating point registers.

Without sorting (top histogram in Figure 5), two major spikes appear in the histogram, one around 60°C, and one around 82°C. When unit usage is constant for a longer period of time, as is the case when many tasks with similar characteristics are scheduled successively, the units reach a steady state temperature (also compare Figure 2). The spike at 60°C results from the floating point registers being inactive during a longer period of time, whereas the spike at 82°C results from the registers being active for a longer period of time.

When runqueue sorting is enabled (bottom histograms in Figure 5), the spikes are diminished, and temperature is biased towards medium temperatures.

The effects of runqueue sorting on temperature become most apparent when overlaying the three histograms, as can be seen in Figure 6. To improve readability, the figure lacks the bars of the histograms and displays only the tops.

As can be seen, runqueue sorting biases temperature towards the middle of the range, which particularly leads to high temperatures occurring less frequently and also reduces the observed maximum temperature. The figure also shows that enhanced sorting guided by a thermal model is superior to simple sorting guided only by unit utilization.

Without runqueue sorting, the temperature of the floating point register file was greater than 80°C for 25% of the time. With simple sorting, this percentage dropped to 9% and with enhanced sorting further to 6%.

For other combinations of tasks, runqueue sorting yielded similar effects: If the tasks possess different characteristics regarding the utilization of a certain unit, the temperature of this units gets biased towards medium temperature ranges by runqueue sorting. Figure 7 shows histograms of the temperature for the data translation look-aside buffer (DTLB), which is the hottest unit when running `gobmk`, a memory intensive application, in combination with `leslie3d`, which is also memory intensive, but does not stress the DTLB as much as `gobmk` does. Although the overall temperature of the DTLB is lower than that of the floating point registers in the previous test, a bias towards the medium range can also be observed. Since `gobmk` and `leslie3d` both use the DTLB to some degree, the spikes in the histogram are not as prominent as in the previous test.

For combinations of tasks that have similar characteristics, runqueue sorting is not beneficial. Figure 8 shows histograms for the temperature of the floating point registers when running `calculix` in combination with `milc`. Both applications show high usage of the floating point registers, although not as high as with `namd`. Therefore, the histogram shows only one spike around 70°C. Since the tasks have similar characteristics, runqueue sorting cannot reduce the temperature of the floating point registers and the histogram looks similar with sorting activated.

We measured the overhead introduced by sorting. We compared our implementation of runqueue sorting to Linux's original scheduling policy. The runtime of the benchmarks increased by 1.3% with simple sorting and by 1.5% with enhanced sorting.

5.3 Activity Balancing

We tested activity balancing by running the same workload described at the beginning of Section 5.2, but on all eight CPUs of the system. Hence, we started 24 instances of `hmmer` and `namd`, which the load balancer distributed to the individual CPUs.

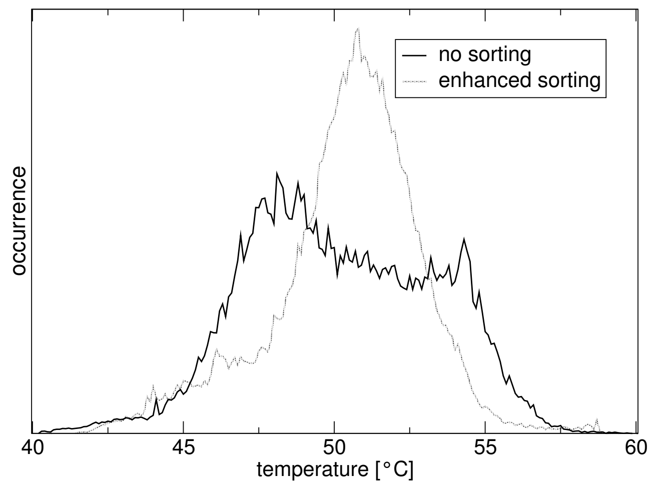

Figure 7: Temperature of dtlb (gobmk + leslie3d)

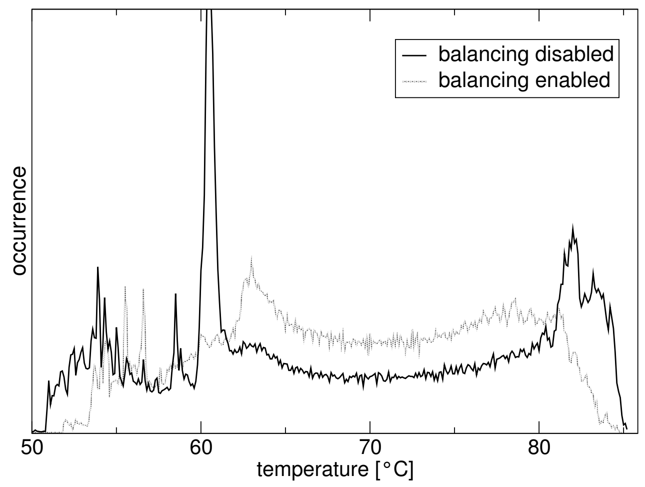

Figure 9: Temperature of floating point registers (8 processors)

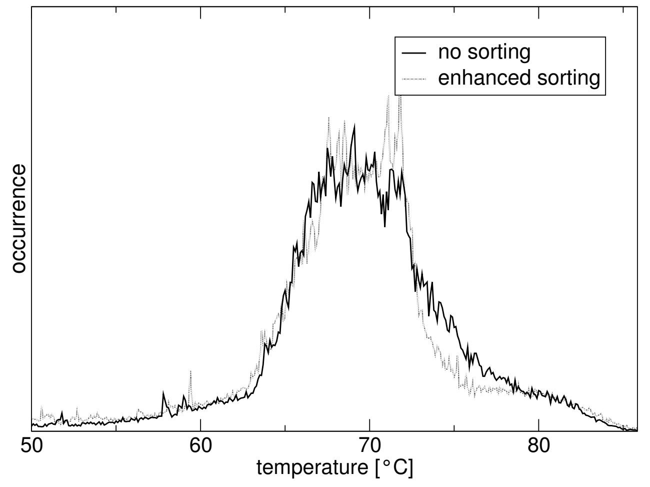

Figure 8: Temperature of floating point registers (calculix + milc)

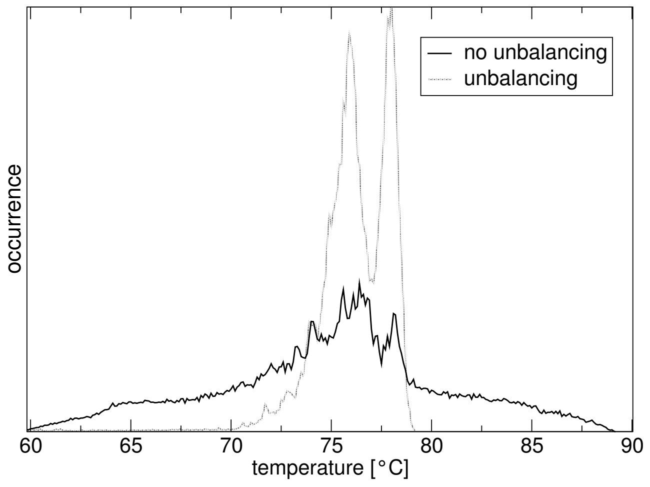

Figure 10: Temperature of floating point registers (HyperThreading)

To eliminate the possibility of buying the reduction of hotspots on one CPU with an aggravation on some other CPU, we observed the temperature of all eight CPUs.

We performed two runs, one with activity balancing disabled and one with activity balancing enabled. During both runs, we activated enhanced runqueue sorting.

Since runqueue sorting depends on runqueues consisting of tasks with different characteristics, it is only beneficial if the load balancer distributes tasks to CPUs accordingly. The default Linux load balancer, however, is oblivious of the tasks' characteristics.

This becomes apparent in Figure 9, which displays histograms of the temperature values accumulated from all eight processors. Without activity balancing, the histogram looks similar to the one in the top half of Figure 5, which resulted from a test run with runqueue sorting disabled. Runqueues containing too many instances of `namd` are responsible for this. Only when runqueue sorting and activity balancing are combined, the desired effect eventuates.

Migrating a task to another processor introduces an overhead, since the task needs to warm up the cache of the new processor. However, activity balancing is an infrequently occurring operation; once the tasks are distributed to the CPUs according to their unit utilization, no further migrations are necessary. We measured only 0.3% of overhead introduced by activity balancing.

5.4 Activity Unbalancing

To test activity unbalancing, we enabled only one CPU in the system, but activated the processor's HyperThreading capability (Intel's term for simultaneous multithreading), which resulted in a system consisting of two logical CPUs. We ran one test with runqueue unbalancing enabled and one test with runqueue unbalancing disabled. Both times, we enabled runqueue sorting and ran four instances of `hmmer` and `namd` respectively.

It is remarkable that with HyperThreading enabled, the histogram (Figure 10) looks completely different than with HyperThreading disabled. Instead of two spikes, one in the low and one in the high temperature ranges (compare Figure 6), with HyperThreading, the temperatures in the middle range dominate.

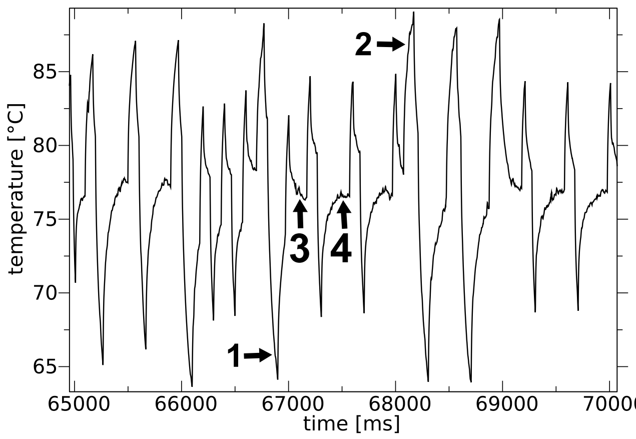

Figure 11: Temperature of floating point registers (HyperThreading)

The reason for this becomes evident when looking at the course of temperature (Figure 11). Power consumption and temperature of the functional units are no longer determined by the order in which tasks are scheduled by one scheduler, but by the scheduling decisions of two independent schedulers. Since task switches happen independently for both logical processors, this leads to situation in which none of the processors uses the functional unit as shown at point (1) in the figure, both processors use the functional unit (2), or one processor uses the functional unit and the other does not (3) and (4).

When one logical processor uses a unit and the other does not, the unit's temperature is in the middle range. Combined with the fact that it takes some time for temperature to increase or decrease after both processors start using or not using a unit at the same time, this yields the bias towards medium-range temperatures that can be observed in the histogram.

Provided that the set of runnable task allows it, activity unbalancing ensures that tasks using complementary resources are always scheduled together on sibling processors. This avoids temperature spikes to the upper and lower ranges (like points (1) and (2) in Figure 11) and ensures that a situation as shown at points (3) and (4), where temperature is in the middle range, dominates most of the time.

The histogram (Figure 10) reflects this: With activity unbalancing enabled, temperature values are much more concentrated in the middle range than without unbalancing. This shows the benefits of activity unbalancing. Without unbalancing, the temperature of the floating point registers is higher than 80°C during 17.7% of the time. Activity unbalancing achieves that the temperature of the floating point register never exceeds 80°C in this scenario, so our policy succeeded in avoiding hotspots.

Activity unbalancing is also advantageous in another respect: Running tasks that use complementary resources together also makes sure that the tasks running simultaneously on sibling processors do obstruct each other less, since they do not compete as much for resources as would be the case when running tasks with similar characteristics together. This is the same principle symbiotic job scheduling [18] takes advantage of. Less obstruction means higher throughput; in our scenario, the runtime of the benchmarks decreased by 3.6% when activity unbalancing was enabled.

We verified that this is generally the case by test runs using other SPEC benchmarks: With one processor and HyperThreading activated, we ran six benchmarks simultaneously. The benchmarks were chosen at random out of the SPEC CPU2006 suite. Whenever one benchmark terminated, we started another benchmark, also selected at random, so there were always six benchmarks running simultaneously, but in arbitrary combinations.

We measured the runtimes of the benchmarks with activity unbalancing disabled and also recorded the order in which the benchmarks were started. After the first run, we enabled activity unbalancing and replayed the same sequence of benchmarks as before. A comparison of the runtimes showed an improvement (reduction of runtime) of 3.3%.

5.5 Analysis

Our experiments show that runqueue sorting improves temperature distribution on the chip and reduces hotspots. Activity balancing creates the prerequisites for runqueue sorting in SMP systems. For multithreaded processors, activity unbalancing reduces hotspots and additionally increases performance.

The overhead introduced by runqueue sorting has to be compared to the overhead that results when throttling is engaged to prevent overheating. The overhead introduced by throttling depends on the actual throttling mechanism, and on the trip temperature at which throttling starts. Both, mechanisms and trip temperatures vary between different CPU types.

As an example, Intel's *thermal monitor 1* feature periodically disables the clock signal, effectively reducing the processor's frequency by 12.5% to 87.5%. The *thermal monitor 2* feature uses frequency scaling for reducing power and temperature; the penalty introduced by thermal monitor 2 depends on the frequency the processor runs with when not throttled. In both cases, throttling results in a performance penalty that is considerably higher than the penalty introduced by our vector-based scheduling policies, which we measured to be 1.8% at most (1.5% introduced by runqueue sorting and an additional 0.3% introduced by activity balancing).

As an example, we showed in Section 5 that for a scenario with tasks using complementary resources, runqueue sorting manages to reduce the percentage of time during which the hottest of the processor's units operates above 80°C from 25% to 6%. If the processor is not supposed to operate at more than 80°C, throttling has to be used to keep temperature below 80°C, at the price of prolonged execution times. If we assume the processor runs at half its normal frequency when throttled, without runqueue sorting, 25% of the instructions have to be executed at half speed, which doubles the time required to process these instructions and increases total execution time by 25%. With runqueue sorting, the total execution time is only increased by 6% because of throttling, plus the additional 1.5% introduced by the overhead runqueue sorting causes. A programm running 100*s* on an unthrottled processor thus runs 125*s* without runqueue sorting and 107.6*s* with runqueue sorting, which is a speedup of 14%. Even if a throttled processor is running at 75% maximum speed instead of 50%, the speedup is still 7%.

Depending on the processor, 80°C, which we use as a reference temperature for illustrating the benefits of our approach, need not necessarily be a critical temperature. In the system we used, we were therefore not able to increase performance by decreasing temperature, since even without our improved temperature distribution, no throttling was engaged. Yet, with increasing power and integration densities, thermal problems can be expected to aggravate in the future. Additionally, the lifetime and the reliability of a processor chip decreases with temperature. Increasing temperature by 10 to

15 degrees halves the lifetime of an electrical circuit [19]. Therefore, reducing hotspots is always beneficial.

6. LIMITATIONS AND FUTURE WORK

The main limitation of our approach lies in its dependence on the workload. For workloads consisting only of tasks with similar characteristics (that means, tasks executing the same kind of instructions), our proposed scheduling policies are not beneficial, and for workloads where each runqueue consists of one task only, they are not applicable. However, recent trends like virtualization and server consolidation can be use to combine more, potentially heterogeneous workloads on a CPU. Therefore, we need to explore to what extent these techniques can be used to create suitable workloads.

A limitation of enhanced runqueue sorting lies in the fact that it requires a thermal model of the processor chip, i.e., the thermal capacitance and resistance of the die, interface material and heat sink must be known. These values differ between different processor types and must be adjusted individually for each system. However, falling back to simple runqueue sorting, where no thermal model is needed, is always an option for systems with unknown thermal characteristics.

Future processors are likely to have built-in mechanisms and policies for avoiding hotspots. For instance, chips that feature spare resources and perform activity migration in hardware have been proposed [8, 17]. Another example are multithreaded chips that favor certain logical processors, preferring them when allocating resources depending on the characteristics of the tasks they run, and on the current temperature distribution [3]. We believe that both hardware and software should cooperate to attain the goal of effective temperature management best. If the hardware exposes information to the operating system about its internal features, like multiple register files or adaptive fetch policies, and about the actions actually taken to prevent thermal emergencies, the operating system can provide the circumstances in which the hardware's mechanisms are most effective, for example by migrating tasks between CPUs or ordering the tasks in the runqueues accordingly.

We believe that our approach is also applicable for multicore processors. In recent multicore processors, throttling can be engaged on a percore basis. Just like for singlecore processors, the decision whether or not to throttle a core of a multicore processor depends on the temperature of the hottest parts of the core, which can be reduced by our proposed scheduling policies. Additionally, having multiple cores on a chip requires higher integration densities, which aggravates thermal problems.

7. RELATED WORK

Characterization of tasks by unit utilization has been used for different purposes in the past.

Isci and Martonosi [11] shows that performance monitoring counters are suitable for determining functional unit utilization by which tasks are characterized. This characterization is used for analyzing the phases a task passes through. Our approach adopts this methodology for characterizing tasks, but uses the characterization for energy-aware scheduling.

Lee and Skadron [13] uses unit utilization inferred from performance monitoring counters to determine the power consumption of the individual functional units on a chip. The energy estimates are used for on-line estimation of the temperature distribution on the chip by means of the HotSpot [9] simulator. We use the methodology proposed by Lee and Skadron to determine the temperature of functional units on the chip. Using the Hotspot Simulator, we were able to verify that our scheduling algorithms succeed at influencing temperature distribution.

Improving performance by taking advantage of the tasks' characteristics and mitigating thermal problems have been the goal of different scheduling policies proposed in the past.

Snavely and Tullsen [18] combines task on a multithreaded processor in a way that results in maximum throughput. In contrast to our approach, thermal aspects are not considered. Also, the optimal combination of tasks is not inferred directly from the tasks' characteristics, but determined empirically by trying out different combinations.

Gomaa et al. [5] combines tasks with different characteristics for running simultaneously on a multithreaded processor to avoid overheating individual units while other units are cold. However, the approach does not characterize tasks by utilization of functional units, but distinguishes only between high and low IPC (instructions per cycle) tasks and between integer and floating point tasks. Also, simulation is used instead of real hardware.

Michaud and Sazeides [15] proposes to use shorter timeslices to prevent overheating individual parts of the chip. We argue that shorter timeslices are only beneficial if there are tasks available that use different units, and that the order in which they are scheduled is important. Our work addresses both problems.

Another approach by Donald and Martonosi [4] proposes task migrations to prevent hotspots in multicore processors. Performance monitoring counters and sensor readings obtained from a simulator are used to characterize tasks, and migrations to another core are initiated when a functional unit reaches a critical temperature. This approach is comparable to our activity balancing policy, which also strives at avoiding hotspots by distributing tasks to CPUs accordingly. However, we use activity balancing to create the preconditions for runqueue sorting, arguing that not only the distribution of tasks, but also the order in which they are executed matters.

Choi et al. [2] proposes scheduling strategies for mitigating thermal problems on multicore processors. The approach characterizes tasks by using the temperature sensors of the Power 5 processor. Although some of the proposed scheduling policies are similar to ours, they are based on a characterization of tasks into hot and cold tasks and consider the location at which energy is dissipated only on a per-core basis.

Approaches that aim at mitigating thermal problems have mostly focused on the hardware level rather than on the operating system. In contrast to the operating system, hardware has no knowledge about tasks, so hardware-related policies operate at the level of hardware threads.

Donald and Martonosi [3] suggests an adaptive fetch policy for multithreaded chips to avoid the formation of hotspots. The approach uses performance monitoring counters to determine the utilization of the integer and the floating point unit for each thread. Depending on the temperatures of these units, the processor prefers the thread using the cooler unit and fetches proportionally more instructions for this thread. Adapting the fetch policy as a hardware level approach could benefit from operating system policies like the ones we propose, ensuring that tasks with different characteristics are available on a processor.

Another hardware related approach to avoiding hotspots consists in adding spare resources such as register files, issue queues, or ALUs on the processor chip and migrating computation to one of those spare resources when the original resource reaches a critical temperature [8, 17]. In contrast to our approach, this requires additional units, which occupy additional die area.

8. CONCLUSION

The task running on a CPU determines temperature distribution and the location of hotspots. Our work addressed the influence the operating system can take on temperature distribution by scheduling tasks in a suitable manner.

We proposed *task activity vectors* as a metric for characterizing a task by the functional units it uses. Our experiments verify that activity vectors are a valuable input for a scheduler that aims at reducing hotspots. We proposed three scheduling strategies that make use of activity vectors:

Runqueue sorting arranges the tasks in a processor's runqueue in a way that tasks using complementary resources are scheduled successively. This reduces thermal stress, since hot units can cool down when a task not using the units is scheduled. *Activity balancing* makes sure that runqueue sorting can be applied reasonably in multiprocessor systems. By distributing tasks with similar characteristics onto all processors of a system, we make sure that the composition of each runqueue is suitable for runqueue sorting. *Activity unbalancing*, in contrast, concentrates tasks with similar characteristics in the same runqueue. Activity unbalancing is beneficial if applied between SMT siblings, ensuring that tasks with different characteristics are running simultaneously, which reduces thermal stress and at the same time increases throughput.

Our experiments show that activity-based scheduling succeeds at reducing hotspots. Our scheduling policies considerably reduce the percentage of time during which the processor operates in high temperature ranges. This alleviates the need for thermal throttling and reduces thermal emergencies. The overhead of activity based scheduling is minimal; in case of multithreaded processors, activity-based scheduling even has negative overhead (increased throughput).

Acknowledgements

We would like to thank Simon Kellner, Raphael Neider, and Jan Stöß, as well as the anonymous reviewers, for their helpful suggestions and comments regarding this paper.

9. REFERENCES

[1] 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. 2003.

[2] J. Choi, C.-Y. Cher, H. Franke, H. Hamann, A. Weger, and P. Bose. Thermal-aware task scheduling at the system software level. In *Proceedings of the 2007 International Symposium on Low-Power Electronics and Design (ISLPED'07)*, Aug. 2007.

[3] J. Donald and M. Martonosi. Leveraging simultaneous multithreading for adaptive thermal control. In *Second Workshop on Temperature-Aware Computer Systems (TACS'05)*, Madison, USA, June 2005.

[4] J. Donald and M. Martonosi. Techniques for multicore thermal management: Classification and new exploration. *SIGARCH Comput. Archit. News*, 34(2):78–88, 2006.

[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):260–270, 2004.

[6] S. H. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, 2001. Q1 issue.

[7] Y. Han, I. Koren, and C. M. Krishna. Temptor: A lightweight runtime temperature monitoring tool using performance counters. In *Proceedings of the Third Workshop on Temperature-Aware Computer Systems (TACS'06)*, June 2006.

[8] 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.

[9] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusamy. Compact thermal modeling for temperature aware design. In *Proceedings of the* 41*st Design Automation Conference (DAC'04)*, 2004.

[10] Intel. *Intel® Pentium® 4 Processor with 512-KB L2 Cache on 0.13 Micron Process Thermal Design Guidelines Design Guide*, Nov. 2002.

[11] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the* 36*th Annual IEEE/ACM International Symposium on Microarchitecture (MIRCO'03)*, pages 93–104, Washington, DC, USA, 2003. IEEE Computer Society.

[12] M. T. Jones. Inside the linux scheduler. *IBM Developer Works*, 2006.

[13] K.-J. Lee and K. Skadron. Using performance counters for runtime temperature sensing in high-performance processors. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 11*, Apr. 2005.

[14] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *First ACM SIGOPS EuroSys Conference*, Leuven, Belgium, Apr. 18–21 2006.

[15] P. Michaud and Y. Sazeides. Scheduling issues on thermally–constrained processors. Technical report, Institut de Recherche en Informatique et Systemes Aleatoires, Oct. 2006.

[16] E. Rothem, J. Hermerding, C. Aviad, and C. Harel. Temperature measurement in the intel core duo processor. In *Proceedings of the Twelfth International Workshop on Thermal Investigations of ICs (THERMINIC'06)*, Aug. 2006.

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

[18] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *SIGPLAN Not.*, 35(11):234–244, 2000.

[19] L.-T. Yeh and R. C. Chu. *Thermal Management of Microelectronic Equipment*. American Society of Mechanical Engineers, 2001.