Dynamic Thermal Management through Task Scheduling*

Jun Yang[†] **Xiuyi Zhou**[†] **Marek Chrobak**[¶] **Youtao Zhang**[§] **Lingling Jin**[‡]

[†]**Electrical and Computer Engineering**
[§]**Computer Science**
University of Pittsburgh, Pittsburgh PA 15261

[¶]**Computer Science**
University of California, Riverside
Riverside, CA 92521

[‡]**Nvidia Corporate**
Santa Clara, CA 95050

Abstract

The evolution of microprocessors has been hindered by their increasing power consumption and the heat generation speed on-die. High temperature impairs the processor's reliability and reduces its lifetime. While hardware level dynamic thermal management (DTM) techniques, such as voltage and frequency scaling, can effectively lower the chip temperature when it surpasses the thermal threshold, they inevitably come at the cost of performance degradation.

We propose an OS level technique that performs thermal-aware job scheduling to reduce the number of thermal trespasses. Our scheduler reduces the amount of hardware DTMs and achieves higher performance while keeping the temperature low. Our methods leverage the natural discrepancies in thermal behavior among different workloads, and schedule them to keep the chip temperature below a given budget. We develop a heuristic algorithm based on the observation that there is a difference in the resulting temperature when a hot and a cool job are executed in a different order. To evaluate our scheduling algorithms, we developed a lightweight runtime temperature monitor to enable informed scheduling decisions. We have implemented our scheduling algorithm and the entire temperature monitoring framework in the Linux kernel. Our proposed scheduler can remove 10.5-73.6% of the hardware DTMs in various combinations of workloads in a medium thermal environment. As a result, the CPU throughput was improved by up to 7.6% (4.1% on average) even under a severe thermal environment.

1 Introduction

As technology for microprocessors enters the nanometer regime, power density has become one of the major constraints to attainable processor performance. High temperatures jeopardize the reliability of the chip and significantly impact its performance. The immense spatial and temporal variation of chip temperature also creates great challenges to cooling and packaging which, for the sake of cost-effectiveness [43], are designed for typical, not worst-case, thermal condition. This entails dynamic thermal managements (DTM) to regulate chip temperature at runtime.

There have been plenty of researches on DTMs at the microarchitecture level [6, 11, 16, 25, 28, 34, 35, 36]. Architecture solutions can respond to thermal crisis rapidly and reduce the chip temperature effectively through various performance reduction mechanisms.

Recently, a number of works have shown great potential in OS-assisted workload scheduling in addition to the hardware level techniques [7, 10, 14, 22, 23, 31]. The main approach is to leverage the temperature variations between different jobs, and swap them at an appropriate time to control the chip temperature. This has been practiced in both CMPs [7, 10, 31] and single-core processors [14, 22, 23]. Our work continues this direction of research.

We develop a heuristic scheduling algorithm to alleviate the thermal pressure of a processor. Our algorithm ThreshHot is based on the observation that, given two jobs, one hot and one cool, executing the hot job before the cool one results in a lower final temperature than after the reversed order. Thus, as long as executing the hot job itself does not violate the thermal threshold, the hot-cold order is better (or, at least, not worse) than the cold-hold order. Consequently, ThreshHot selects at each step the hottest job that does not exceed the thermal threshold.

ThreshHot outperforms other scheduling algorithms such as the one that changes the priority ranks of the hot and the cool jobs [22]. To know which job will be hot or cool for the hotspot, we develop a highly efficient *on-line* temperature estimator leveraging the performance counter based power estimation [19, 20, 22], compact thermal modeling [35], and a fast temperature solver [12]. We implemented the estimator for a Pentium 4 processor, although our general methodology is applicable to other processors such as CMPs. We calibrate and validate the model parameters against real measurements on our processor package. We also implemented our scheduling heuristics in the Linux kernel, together with our temperature estimator, and we tested the entire framework over the complete executions of SPEC CPU2K benchmarks, mediabench, packetbench and netbench. ThreshHot can remove up to 73.6% (34.5% on average) hardware DTMs in a medium thermal environment. With all the context switching, temperature estimation, and the thermal-aware scheduling overheads considered, ThreshHot consistently improves the performances of a mix of hot and cool programs by up to 7.2% (4.7% on average) compared to a base case with traditional thermal-oblivious Linux task scheduling. Our scheduling algorithm targets only batch jobs and thus has unnoticeable impact on interactive jobs and no impact on real-time applications.

The remainder of the paper is organized as follows. Section 2 discusses previous related works. Section 3 elaborates on our thermal-aware heuristic algorithm through mathematical derivations. Section 4 explains how to obtain online power and thermal information for our scheduler to work properly. Section 5 introduces our modifications of the Linux kernel scheduler. Section 6 compares our proposed scheduler with other alternatives. Section 7 reports the experimental results comparing ThreshHot to three other algorithms. Section 8 concludes this paper.

2 Prior Work

Some recent works have developed temperature control techniques for regular [32] and *real-time* [1, 2, 39, 40] workloads. The main approach is to dynamically adjust the CPU speed to minimize the peak temperature of the CPU, subject to the constraint that all jobs finish by their deadlines. Similar ap-

*This work is supported in part by NSF grants CCF-0734339, CNS-0720595, OISE-0340752 and CCF-0641177.

proaches can be used to minimize the energy consumption for real-time systems [30, 41]. Temperature control through job scheduling has also been utilized to enhance the reliability of a processor [26]. In contrast, our objective is to maximize the performance by scheduling the workloads to keep the temperature below a given threshold. Note that the threshold can be the manufacturer-defined temperature threshold[1] for the physical chip, or an OS-defined threshold for a system to stay within a thermal envelope. Hence, we always attempt to run workloads with full speed as long as the temperature stays below the given threshold.

Thermal management through workload scheduling has been studied in various scenarios. In CMPs, the "Heat-and-Run" technique performs thread assignment and migration to *balance* the chip temperature at runtime [31]. In another work [10], a suite of DTM techniques, job migration policies, and control granularity are jointly investigated to achieve the maximum chip throughput. Also recently, a simple periodic thread swapping between two cores to balance the chip temperature was studied on a dual-core processor [7]. All these approaches exploit simple interleaving between hot and cool jobs to improve thermal characteristics of a schedule. Our objective is to find the best thread for a core when it becomes hot, and this thread may not be the coolest available thread. For example, when there is both a medium hot and a cool thread, our scheduler will pick a medium hot thread as long as it will not trigger DTM. In this paper, we demonstrate this philosophy using a scheduling heuristic on a single-core processor, and leave its extensions to CMPs as future work.

In single-core domain, the "HybDTM" [22] controls temperature by limiting the execution of the hot job once it enters an alarm region. This is achieved by lowering the priority of the hot job so that the OS allocates it with fewer time slices to reduce the processor temperature. The same principle can be seen in [4], where energy dissipation rate is evened among hot and cool jobs through assigning different CPU time to them. Our technique does not modify the time allocated to hot and cool jobs, as this would affect the fairness policy of the system. Instead, we attempt to rearrange their execution order within each OS epoch to lower the overall temperature. This allows us to control the temperature while preserving priorities among different jobs.

Thermal control through workload management has also been studied at the system level. In [29], a temperature-aware workload placement heuristic was studied for data centers to minimize the cost of cooling. The "Mercury and Freon" [15] framework uses a software to estimate temperatures for a server cluster and manages its component temperatures through a thermal-aware load balancer. The "ThermoStat" [8] tool employs a detailed 3D computational fluid dynamics model for a rack-mounted server system. This tool can guide the design of better dynamic thermal management techniques for server racks. Our work targets at CPU temperature control, which can be complementary to system level thermal management schemes.

3 Thermal Scheduling Algorithms

When the processor overheated and forced to slow down, nearly all vital measures will be degraded: throughput and utilization will be reduced, response time will increase, jobs are more likely to miss deadlines, etc. Thus, independently of the characteristics and focus of a given system, processor overheating will negatively affect its performance.

When incorporating new features, such as thermal awareness, into a scheduler, it is desirable to make them as transparent to the user as possible; in particular, to keep the existing scheduler structure and properties. For this reason, we focus our work on a batch system for which the main objectives are the minimum turnaround time, maximum throughput, and CPU utilization. For batch jobs, the OS periodically interrupts the job execution to maintain its statistics and determines if a different job should be swapped in and, if so, which one. We amend the decision of which job should be selected next with thermal-awareness while keeping all other features intact. Therefore, in every scheduling interval, the OS needs to select the best job anticipating that such a selection would lead to the overall least amount of thermal violations.

3.1 The Principle

To keep the temperature below the threshold, the naïve, greedy approach would be to minimize the *current* chip temperature by executing at each step the coolest available job. As a result, the jobs are scheduled in the order of increasing temperature, from coolest to hottest. As it turns out, however, the greedy schedule actually increases the chances of exceeding the temperature threshold in the long run. To see this, consider a simple case where at some schedule interval t only two jobs x and y are available, with power consumption P_x and P_y respectively, where $P_x < P_y$ (so x is cooler than y). We will show that if we execute these jobs in order xy (x before y, as in the greedy schedule) then the temperature at the end of $t+1$ is higher than for the order yx (y before x). This means that **if the temperatures for both orders stay below the threshold**, then order yx is less likely to cause a DTM in the future.

Consider the simplified thermal model for a processor treated as a single node. The duality between heat transfer and electrical phenomena [21] has provided a convenient basis for modeling the chip temperature using a *dynamic compact thermal model* [35]:

$$\frac{1}{R}T + C\frac{dT}{dt} = P, \tag{1}$$

where T is the temperature relative to the ambient air, R and C are, respectively, the chip's effective vertical thermal resistor and capacitor, and P is the power consumption. Note that when $\frac{dT}{dt} = 0$, the chip reaches its steady temperature RP which depends on the average power of a job. The time to reach the steady temperature is determined by the RC constant ($R \times C$) of the thermal circuit. However, when the chip is switching among different jobs prior to the steady temperature, it is always in a transient stage (i.e. $\frac{dT}{dt} \neq 0$).

Discretizing the time scale into small time steps Δt and denoting by T_i the temperature at time $i\Delta t$, Equation (1) can be approximated by

$$\frac{1}{R}T_i + C\frac{T_i - T_{i-1}}{\Delta t} = P. \tag{2}$$

Rearranging the terms, we have $T_i = \alpha T_{i-1} + \beta P$, where $\alpha = \frac{RC}{\Delta t + RC}$ and $\beta = \frac{R\Delta t}{\Delta t + RC}$ are constants dependent on Δt and, clearly, $\alpha < 1$. If each scheduling interval is divided into n steps of length Δt, the temperature at the end of this interval can be expressed as:

$$T_n = \alpha^n T_0 + (\alpha^{n-1} + \alpha^{n-2} + \cdots + 1)\beta P. \tag{3}$$

[1]This threshold is a safe operating temperature beyond which the chip might be damaged due to overheating, and exceeding it triggers DTMs.

For schedule xy, the temperature after completing y ($2n$ steps) will be

$$T_{2n}^{xy} = \alpha^{2n} T_0 + (\alpha^{n-1} + \alpha^{n-2} + \cdots + 1)\beta(\alpha^n P_x + P_y), \quad (4)$$

while for schedule yx, this final temperature will be

$$T_{2n}^{yx} = \alpha^{2n} T_0 + (\alpha^{n-1} + \alpha^{n-2} + \cdots + 1)\beta(\alpha^n P_y + P_x). \quad (5)$$

It is now easy to see that $P_x < P_y$ implies $T_{2n}^{yx} < T_{2n}^{xy}$. That is, *scheduling the hotter job first results in a lower final temperature*. We emphasize that this is benecitial **only when running the hotter job first does not increase the temperature above the threshold**. Figure 1 gives an intuitive illustration of the impact of scheduling on temperature. The graph shows temperature variation for the IntReg unit with two different power inputs, representing two different jobs. They are scheduled in two different orders as just described. The graph was obtained using a full-chip thermal model (rather than a single node as a whole) solved by the fourth order Runge-Kutta method. As we can see, running the hotter job first results in lower final temperature. If the chip's thermal threshold is in between the difference of the two ending temperatures, the greedy schedule would cause a thermal violation.

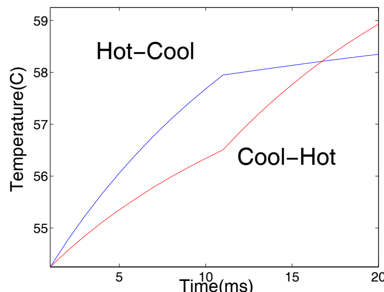


Figure 1. The impact of scheduling a hot and cool program in different orders.

Suppose now you are given a schedule for some number of job intervals. Suppose further that in this schedule there are two consecutive job intervals x, y with x before y, such that $P_x < P_y$ and that executing y first will not exceed the threshold. Then, by the reasoning above, we can exchange x with y, and this swap will not increase the number of thermal violations in the whole schedule. The reason is that in this new schedule, after completing yx, the temperature will be lower than in the original schedule after completing xy, so we cannot cause an increase of the temperature at any given step later in the schedule. This observation naturally leads to the following heuristic:

$\mathcal{P}$: *at each step choose the hottest job* ***that will not increase the temperature above the threshold***.

The above policy $\mathcal{P}$ is the basis of our algorithm ThreshHot. We emphasize that $\mathcal{P}$ does not guarantee to minimize the total number of thermal violations for a given set of job intervals (in fact, in a more rigorous setting, this problem can be shown to be NP-hard [9]); nevertheless, it computes a local minimum and it constitutes a reasonable heuristic for our application.

We also need to address the case when no job interval satisfies policy $\mathcal{P}$, i.e. all the jobs would increase the temperature above the threshold. In this case, it is most beneficial to pick the *hottest* job interval for execution. This is because the hardware thermal management (e.g. DVFS) will be triggered to cool the chip regardless of which job we choose, and selecting the hottest job interval at this time reduces the likelihood of a future thermal violation.

For example, suppose there are three job intervals available, say J_1, J_2 and J_3 with descending powers. If picking J_1 would increase the temperature above the threshold while picking J_2 would not, then policy $\mathcal{P}$ will first pick J_2 to run. If all of them would exceed the threshold, $\mathcal{P}$ will pick J_1.

We remark here that the OS fairness policy imposes some restrictions on how long a job interval can be postponed (this will be discussed in more depth in Section 5). Thus, in addition to the rules described above, the choice of the next job to run must be consistent with these fairness restrictions.

3.2 In Practice

In the earlier discussion we assumed a simple case where the CPU is considered as a single node and the heat is only dissipated through the vertical thermal resistor and capacitor. In reality, there is a great temperature variation on-die and only the temperature at the hottest spot should be maintained below the threshold. This scenario is more complex than for a single node, as the heat can also be dissipated laterally. Therefore, the thermal model in Equation (1) will be expanded into a matrix form in which every node is described by:

$$\frac{T - T_1}{R_{L1}} + \frac{T - T_2}{R_{L2}} + \frac{T - T_3}{R_{L3}} + \frac{T - T_4}{R_{L4}} + \frac{T}{R} + C\frac{dT}{dt} = P \quad (6)$$

where the first four extra terms describe the heat transfer from the central node (with temperature T) to its lateral neighbor nodes (with temperature T_1-T_4). The number of neighbors per node depends on the processor floorplan and how the system is discretized. We have shown four nodes as an example, with T_i being the temperature for the ith neighbor node, and R_{Li} being its lateral resistance from the central node.

The temperature T of the hottest spot on-chip, described by Equation (6), is higher than the T_i's. Also, heat is removed mostly from the vertical path and less from the surface [10, 31, 35]. In more quantitative terms, our experience with a full-chip model shows that the R_{Li}'s are typically 10$\sim$20 times the R for a hot unit such as the IntReg. The resulting lateral RC time constants are on the order of 100 milliseconds and vertical RC time constant is less than 10 milliseconds. Since the left hand side of Equation (6) is dominated by the last two terms, we can still treat a hotspot as a single node as before.

4 Obtaining Power and Temperature Online

As discussed above, our thermal-aware scheduling algorithm needs information about the peak temperature of the processor and power usage for the executed jobs. In this section, we explain how these values can be computed at runtime.

4.1 Computing the Temperatures

Most current processors are equipped with an on-chip thermal sensor for detecting the chip temperature at runtime. The sensor compares the current temperature with a preset threshold and signals a violation if the former is higher. The hardware then responds to such a signal immediately by throttling the performance so that the chip generates less power and, as a result, the temperature starts to drop. In Pentium 4, for example, the performance is throttled by dynamic frequency scaling — the

frequency is scaled by half until the temperature drops below the safe threshold [43].

Thermal sensor readings are insufficient. It seems that the OS could leverage such on-chip thermal sensors for temperature readings. Unfortunately, this is insufficient because, in addition to the current temperature, our algorithm also needs the temperature in the *next* time interval. Further, for a job not currently in execution it is difficult to determine, from its temperature history, what its temperature might be in the future. For example, suppose a job was swapped out last time at 65°C, and currently the sensor reading is 60°C. The temperature for this job in the next time interval may be either higher or lower than 60°C. This is because the future temperature depends on several factors: the current temperature, the power consumption of this job in the next time interval, and the length of the next interval.

Temperature model. Formally, $T_{next} = F(P, T_{current}, \Delta t)$ where P is the average power in the next interval, Δt is the interval length, and function F is characterized by:

$$\mathbf{G}T + \mathbf{C}\frac{dT}{dt} = P \,, \tag{7}$$

which is the matrix form of Equation (1) with $\mathbf{G}$ being the matrix of the thermal conductances. Both T and P are now vectors. Each element corresponds to one modeling node. Therefore, to obtain the temperatures in the next time interval for a candidate job interval, the scheduler must solve Equation (7) from $T_{current}$ (which can be read from sensors), P of the job (which can be projected from its past power consumption), and Δt (which is a fixed value). The sensor readings alone cannot lead to a quantitative comparison with the threshold.

Temperature calculation. This may seem like a lot of computation for the scheduler to solve (7) at runtime. Fortunately, previous work has shown that the complexity of Equation (7) can be greatly reduced if the time interval Δt is kept constant [12]. This is the case in our scheduler. Here, we briefly discuss our temperature estimation method.

The linear system in equation (7) has a complete solution as:

$$T(t) = e^{\mathbf{C}^{-1}\mathbf{G}t}T(0) + \int_0^t e^{\mathbf{C}^{-1}\mathbf{G}(t-\tau)}\mathbf{C}^{-1}P(\tau)d\tau \tag{8}$$

For a fixed-length scheduling interval Δt, we take the average power during the interval so that $P(t)$ can be factored out. (8) is now:

$$T(\Delta t) = \mathbf{A}T(0) + \mathbf{B}P \tag{9}$$

where $\mathbf{A} = e^{\mathbf{C}^{-1}\mathbf{G}\Delta t}$, and $\mathbf{B} = \int_0^{\Delta t} e^{\mathbf{C}^{-1}\mathbf{G}(t-\tau)}\mathbf{C}^{-1}d\tau$. Both $\mathbf{A}$ and $\mathbf{B}$ are constant matrices with a constant Δt. Since linear system (9) is time-invariant, it holds for every interval Δt. Therefore:

$$\begin{array}{l} T(n\Delta t) = \mathbf{A}T((n-1)\Delta t) + \mathbf{B}P(n-1), \quad \text{or simply} \\ T(n) = \mathbf{A}T(n-1) + \mathbf{B}P(n-1) \end{array} \tag{10}$$

As we can see, once $\mathbf{A}$ and $\mathbf{B}$ are pre-calculated and stored, temperature at any step n can be found through linear combination of the temperature and power at step $n-1$. When used online, $T(n-1)$ is the current temperature, $P(n-1)$ is the power dissipated by a job in the next scheduling interval, and $T(n)$ is the temperature at the end of the next interval. Computing the $T(n)$ now is very inexpensive. Our thermal model has 82 nodes in total and computing the 82×1 temperature vector at runtime takes only $\sim$16.45μs. Next, we will discuss how to obtain the power values $P(n-1)$ online.

4.2 Computing the Powers

Power estimation. Recent research has proposed to incorporate on-chip power sensors for power and thermal control [27]. With on-chip power sensors, the OS can obtain the runtime power consumption of critical components easily and quickly. Though such technology is not readily available, some other alternatives have been proposed before and were demonstrated to be very fast and effective. We adopt the method that uses the performance counters to monitor runtime power consumption [3, 19, 20]. Counters provided by high-performance processors such as the Pentium and UltraSPARC can be queried at runtime to derive the activities of each functional unit (FU). When combined with FUs' per access power, their dynamic power and the total chip power can be obtained. However, earlier works either did not consider the *leakage power* or used a constant as a proxy, since leakage is dependent on temperature, which was difficult to obtain at runtime. When the processor runs at a high temperature, its leakage can contribute significantly to the total power [18]. Since we also calculate the temperature online, we consider the leakage as an integral part in our power estimation. We adopted a model developed in [13, 24] using PTM 0.13μ technology parameters [44], matching our processor technology size (Pentium 4 Northwood). We determined the necessary device constants through SPICE simulation.

Power prediction. The last issue we need to resolve now is the prediction of power consumption of a job in the *next* scheduling interval, as required by Equation (10). Here, we face a tradeoff between complexity and accuracy, for a high quality predictor would typically require large memory to store the history information and significant computation time for processing this information. Table-based schemes are likely not appropriate for our framework, for the kernel has a strict limit on the memory space for storing the control information of each job. For example, a good hash table based power predictor that we considered exceeded the kernel space limit, and a small fully-associative table predictor could slow down the program by ∼6%. Therefore we settled for the simple – but cost-effective and fast – *last-value-based* predictor which always uses the last power values to predict those in the next interval. Its error rates for our experimented benchmarks, including 22 SPEC2K, 4 mediabench, 10 netbench, and 4 packetbench, are shown in Figure 2. As we can see, on most programs it has less than 10% error rate. High misprediction rates are seen in *bzip, jpegenc, jpegdec, crc,* and *md5*. Our experiments with those programs (in Section 7) did not show significant disadvantages in most cases, indicating that (at least in those cases) mispredictions did not lead to much mis-scheduling. This is easy to explain: in order for a major mis-scheduling to occur, the prediction error would have to be large enough to either change the jobs' relative temperature ranking, not only their numerical vales, or to incorrectly predict a thermal violation. With moderate prediction errors, the relative ranking of jobs with very different power values will likely remain the same, while for jobs with similar power values, the negative effects of mis-scheduling are small. This is confirmed by our results for *crc* and *md5*, both of which tend to alternate between two different power levels. Here, the last-value predictor often missed the right value but, since the error does not lead to big temperature changes, this did not impact the scheduling decision.

Figure 2. Average error rates for last power value predictor.

4.3 Workflow Summary

To summarize, at the end of each scheduling interval, the OS probes the performance counters from the processor. Those counters record the activities of the current job during the past interval. They are then converted into the power consumption values at the granularity of functional units. Power prediction is performed at this time. The past power values are then fed into a full-chip thermal model for computing the current temperature at the current scheduling interval. For all candidate jobs, their future temperatures are also calculated at this time using their predicted power values. All those future temperatures are sent to the scheduler to determine the next job selection. The flow is depicted in Figure 3(a).

(a) Without thermal and power sensors

(b) With thermal and power sensors

Figure 3. Thermal-aware task scheduling methodologies.

Alternatively, if the processor has available thermal and power sensors, the OS can directly read information from the sensors to compute the future temperatures, as illustrated in Figure 3(b). However, this would entail *many* sensors as the future temperature calculation needs fine-grained power and temperature information. If the sensors are very few, probing the counters is still necessary but the sensor readings can be used for on-line self-calibration to lessen the error due to thermal and power model abstraction.

5 Linux Kernel Implementation

To evaluate our thermal-aware scheduling policy, we implemented all the modules in Figure 3(a) into a Linux kernel version 2.4.18 with O(1) scheduler patch. The major challenge is to insert the new scheduling policy into the existing scheduler while retaining its features. We will first introduce briefly the mechanism of the Linux scheduling [5] and then describe our modification.

5.1 The Skeleton of the Linux Scheduler

The Linux OS distinguishes three classes of jobs: interactive jobs, batch jobs and real-time jobs. The real-time jobs are given the highest priorities while the other two are initialized with the same default priorities. Based on different priorities, the kernel assigns each job a "time quantum". High-priority jobs are given larger time quantum than low-priority jobs. At runtime, all jobs are put into their corresponding "priority queues", and then selected for execution in a descending priority order. Each job occupies the CPU for its allocated time quantum, unless a certain event triggers a swapping, e.g. an I/O request. When a job uses up its time quantum, it is moved into an "expire queue" and the scheduler selects the next job to run. When all the jobs finished using their assigned quanta, an "epoch" is completed. All jobs in the expire queue are now assigned new time quanta determined from their priorities – and a new epoch starts.

5.2 Our Modification

The execution of a time quantum is periodically interrupted by the kernel's interrupt handler, typically once every 1-10*ms*. This is the time when a context switch may happen. We choose to insert our scheduling in this interrupt handler to force a context switch on every *thermal scheduling interval*.

Scheduling interval length. First, we need to decide on the length of scheduling intervals. Since our objective is to keep the peak temperature below the threshold, our scheduling interval should not be much longer than the RC constant of the hottest unit. Previous works assumed $10ms$ as the RC constant of the hottest unit on a CMP processor [10, 31]. From our own experience, we found that the vertical RC constant for the hottest unit is around $7ms$ while the lateral RC constant is on the order of $100ms$. Due to certain implementation requirement (the counter rotation effect [38]), we chose the thermal scheduling interval to be $8ms$. Thus, if the default interrupt frequency is once every $2ms$(or $1ms$), we might force a context switch on every 4(or 8) interrupts.

Context-switch overhead. In the original scheduler, jobs can occupy the processor for its entire time quantum. For batch jobs, the default time quantum is $100ms$ [5]. With an $8ms$ swapping frequency, we could have increased the number of context switches by 12.5 times. We measured the absolute time for each context switch to be $\sim$35.35μs on average. Hence, the context-switch overhead on an $8ms$ interval is 0.044%. Most importantly, as we will show in our final experiments, the thermal-aware scheduler does not necessarily switch to a different job every $8ms$ – quite often the current job is run again. This alleviates the potential for increasing the total number of context switches – without lessening the benefits of our scheduling policy.

Fairness. In the original scheduler, the new epoch does not begin until all the jobs have finished their assigned quantum. When we enforce the thermal scheduling every $8ms$, every quantum is effectively further divided into smaller slices and these slices are executed following our scheduling policy. Therefore, a slice may be delayed due to its potential high temperature, but will not be postponed beyond an epoch. All slices will eventually be executed since they all belong to certain quantum. This is guaranteed by the original scheduler.

Impact on non-batch jobs. Recall that we apply our thermal-aware policy only to batch jobs; but we still need to consider possible indirect impacts on real-time and interactive jobs. Batch jobs are given different range of priorities than the real-time jobs. The candidate jobs that are eligible for thermal scheduling fall within the batch job's priority range. This ensures that we do not touch any real-time jobs and they are scheduled in the same way as before. For interactive jobs, there is

no easy way to distinguish between them and batch jobs. Linux implements a sophisticated heuristic algorithm based on the past behavior of the job to decide whether a job should be considered as interactive or batch. We experimented with a GUI application `VNCplay` developed by Zeldovich *et al.* [42], and we observed that the user response time change due to thermal scheduling is not perceptible.

6 Anatomy and Comparison of Different Scheduling Algorithms

With proper implementation in the Linux kernel, we are now ready to examine the effectiveness of our proposed scheduling algorithm, compared against several alternatives. To show the distinctions among different algorithms, we created three programs that are hot (computation intensive), warm (medium computation and memory accesses), and cool (memory intensive), respectively. We then tested the following scheduling algorithms on the mix of three jobs:

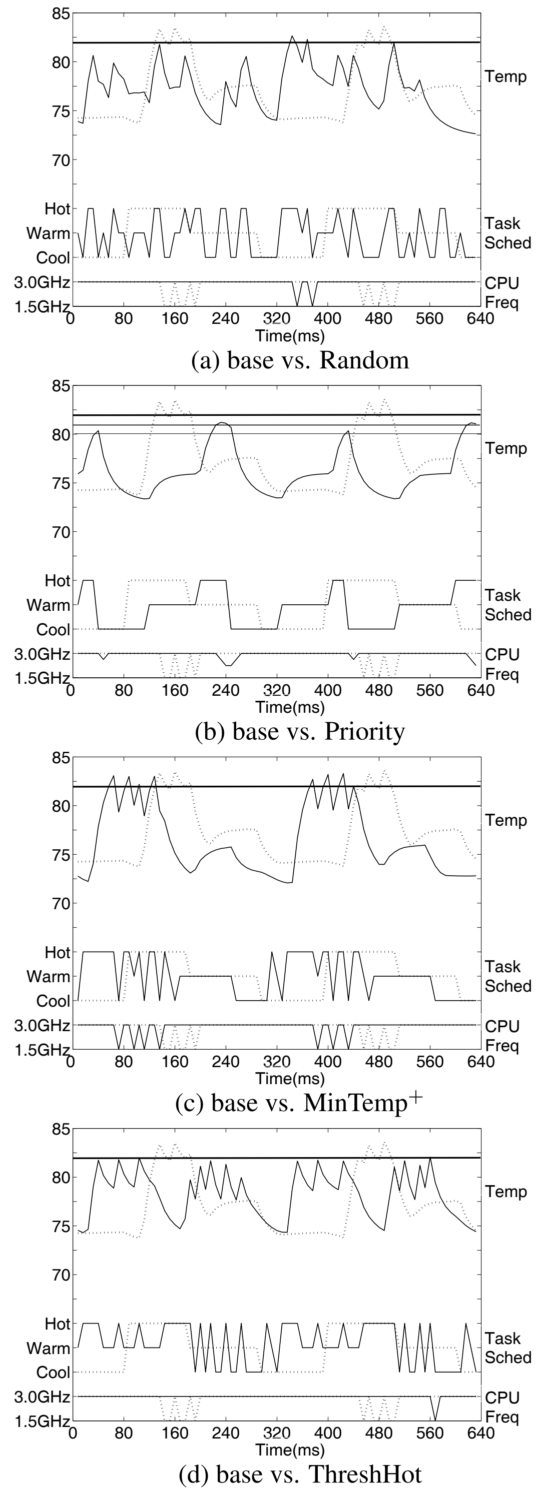
1. Random — This algorithm randomly selects a job to execute in every scheduling interval ($8ms$). We test this scheduler to measure whether the performance improvements can be attributed simply to frequent context switches. This helps to show how much more effective a guided job selection can be in controlling the temperature.

2. Priority — This algorithm lowers the priority of the hot jobs and raises the priority of the cool jobs for every new epoch [22]. A job is considered "hot" if its overall temperature in an epoch exceeds a pre-defined soft threshold which is lower than but close to the hardware threshold. The priority is adjusted proportionally to the proximity of the job's temperature to the hardware threshold. Since the time quanta are calculated based on priorities, this scheduler in effect allocates less CPU time to hot jobs and more to cool jobs within an epoch.

3. MinTemp$^+$ — This algorithm selects the coolest job if its temperature is over the threshold, and selects the hottest job if the current temperature is below the threshold [23]. We improved the original design of MinTemp in that we select the "hot" or "cool" slices based on the jobs' *transient temperatures*, as opposed to their *steady temperatures* (the global temperature trends of programs). Using steady temperatures could produce significant errors as 1) there are often great temperature variations within jobs (Figure 5 shows this property), and 2) even thermally stable jobs will be mostly in their *transient* state when they are constantly swapped in and out. Our improvement can clearly discern temporarily cool slices in a hot job and temporarily hot slices in a cool job, hence, helps the scheduler follow the policy correctly.

4. ThreshHot — This is our proposed algorithm. It selects the hottest program that does not increase the temperature above the threshold. If such job does not exist, it selects the hottest job to run.

Figure 4 shows the execution details of three different jobs under the default Linux scheduler (our baseline scheduler) and the above four schedulers. For clarity, two epochs are shown and all graphs have the same baseline scheduling results so that the differences among the four thermal-aware algorithms are evident. When executing the mix of the three jobs, the baseline thermal-oblivious scheduler picks the job in an ad-hoc manner: in this case cool, hot and warm. The resulting temperature increases above the threshold three times per epoch. This can



(a) base vs. Random

(b) base vs. Priority

(c) base vs. MinTemp$^+$

(d) base vs. ThreshHot

Figure 4. A close-up of the execution traces for four different algorithms. Each graph compares the default Linux scheduler (dashed line) with one algorithm (solid line). In all graphs, the top portion shows the temperature variation with time. The middle portion shows the job switching sequence and the bottom portion shows whether a frequency scaling, a reduction from 3GHz to 1.5GHz (downward arrow), occurred.

been seen from the three downward arrows (drops from 3GHz to 1.5GHz) in the bottom part of the graphs. The three thermal violations happened after the hot job ran for a while. We now compare and contrast how the other four schedulers impact the peak temperatures.

Random scheduler. As we can see from Figure 4(a), the Random scheduler switches to a different job, randomly picked from the job pool, on every scheduling interval. The advantage is that it may select a hot job to run while the chip is cool, and vice versa. This can be seen from the beginning of the first epoch — the base scheduler runs the cool job continuously, while the Random scheduler swaps among all three different jobs, giving the hot job some opportunities to run at a low temperature. Such randomness can remove some frequency scalings when the hot slices are scattered, e.g. in the first epoch, but cannot prevent the scalings judiciously if the hot slices happen to run back-to-back, as with the beginning of the second epoch.

Priority scheduler. This scheduler regulates temperature through adjusting job priorities to allocate less CPU time to hot jobs and more to cool jobs. The granularity of this scheduler is more coarse than that in those discussed earlier since priorities can only be changed between epochs. As a result, the temperature does not respond immediately to the change of priorities. More importantly, since hot jobs are executed less frequently than cool jobs, the cool jobs are likely to finish earlier than the hot jobs. As we can see from Figure 4(b), the schedule of jobs has similar shape as the baseline except that the hot job slices are much shorter (and each epoch is shorter as well). This essentially puts off the execution of hot jobs, which may trigger significant frequency scalings when the cool jobs are exhausted. As we will show later, this is the main reason for this scheduler to fall behind the base scheduler.

The original scheduler also employed two additional thresholds for increasing frequency scaling strengths, as shown in the figure. The hardware control takes two steps to gradually increase the frequency scaling factor (via programming a hardware register) before the temperature reaches the absolute emergency point. This is why the peaks in the temperature curve are smoother than the baseline, and also why the downward arrows at the bottom do not reach 1.5GHz. While this can help to prevent thermal emergencies, it does not prevent frequency scalings. In fact, the frequency scaling may happen more often, though at a lower strength, because the temperature may reach the lower thresholds but not the highest one, as shown in the first frequency dip in the figure.

MinTemp$^+$ scheduler. This scheduler tends to oscillate between the hottest and the coolest job, as shown in Figure 4(c). As we can see, at the beginning of an epoch when temperature is low, the hot job is selected for execution. It runs for some time until a thermal violation occurs. At this point, frequency scaling is engaged *and* the cool job is swapped in. The temperature reduces quickly below the threshold until the end of the window, at which point the hot job is immediately swapped in again. We notice that the cool job is swapped in during frequency scaling, thus, being unfairly penalized for thermal violations caused by the hot job. We will show in Section 7 that the hot job can be sped up while the cool job can be severely punished. On the other hand, when cool jobs are swapped in during a frequency scaling, the processor cools down more quickly than in the base scheduler. This can help to reduce the average temperature when it is close to the threshold, as we can see from the figure. As we will show later, this algorithm can reduce the number of frequency scalings by a moderate amount.

ThreshHot scheduler. In contrast to MinTemp$^+$, our ThreshHot algorithm first estimates the temperatures for all jobs in the next time window and then selects the hottest job that will not exceed the threshold (according to the estimates). Hence, at the beginning of an epoch in Figure 4(d), the hot job is selected to run until the temperature is too close to the threshold. At this point, the scheduler decides to discontinue the hot job and swap in the warm job because it predicts that the warm job will not create a thermal violation in the next interval. The warm job now will run for several intervals until the temperature is low enough for running another hot job slice. As we can see from the figure, at the beginning of each epoch, the scheduler toggles between the hot and the warm job, allocating longer duration for the latter (as opposed to switching between the hot and cool job in MinTemp$^+$). Later in the epoch, warm job's quantum is used up, so the scheduler toggles between the hot and the cool job with longer duration allocated to the latter as well. Such a scheme effectively keeps the temperature right below the threshold achieving the least amount of frequency scaling. For the two epochs shown in the figures, the ThreshHot scheduling shows that it is possible to greatly reduce or even avoid frequency scaling if the jobs are arranged in a good order.

In summary, all schedulers try to keep the temperature below the threshold. The Random scheduler takes an opportunistic approach to disperse hot slices in each epoch. As we will show in our experimental results, there is still much room for improvement if the job selection is well-guided. Priority and MinTemp$^+$ take a more *indirect* approach by lowering the average power locally using the cool job's intervals. However, both cannot avoid the high average power when the cool job's intervals are exhausted. ThreshHot takes a more *direct* approach by picking the job order to regulate the temperature just below the threshold, at the minimum "expense" of cool jobs. These cool jobs are thus "saved" for the future, as precious cooling resources. In contrast, Priority or MinTemp, once the cool jobs are exhausted, will fall back to a baseline thermal-oblivious scheduler.

7 Experimental Evaluation

Unlike in some previous works, in our thermal-aware scheduling the temperature control is not only a goal in itself, but also a tool for improving performance. Such improvements are possible, because fewer thermal violations reduce the number of frequency scalings (or other DTMs). We performed quantitative measurements on the performance with and without thermal-aware scheduling, on a Linux machine using a Pentium 4 Northwood core as our test processor. The core comes with performance counters that are accessible from the kernel. The thermal model was adapted from the HotSpot3.0 toolset [17, 18, 35] with the Pentium 4 floorplan. The DTM used by Pentium 4 is clock throttling which is equivalent to frequency scaling, but with less overhead. We remark that our scheduler will work for any other forms of DTM such as DVS (dynamic voltage scaling).

7.1 Thermal Model Calibration

In the online temperature calculation described by Equation (10), the most important part is to determine the entries in the (constant) matrices **A** and **B**. All these values depend on the processor- and package-dependent thermal RC. From our expe-

rience, even a small variation in certain R and C values can lead to a significant deviation in temperature. Therefore, in order to accurately approximate the program's temperature during execution, it is vital to carefully calibrate our model's parameters.

We performed four real measurements on the processor package — three point measures at three different layers, and one measure in ambient air — for calibrating the RC values: 1) An on-chip thermal diode reading; 2) A thermometer reading on the heat spreader; 3) A thermometer reading on the heat sink; and 4) A thermometer reading for the ambient air. We then proceed to calibrating the RC values in order to match the simulation outputs with the real measurements. Our objective is to minimize the squared error summed over all the programs we measured. This is defined as:

$$error = \sum_{all\ programs} |T_{measured} - T_{simulated}(x_1, x_2, \cdots)|^2$$

where x_is are our parameters to be adjusted. This is a minimization problem that can be tackled by the *Conjugate Gradient* (CG) method [37] which is an algorithm for finding the nearest minimum of a function of n variables. We repeated the CG a number of times. Each time we add a random offset to the computed result and start the next round. The final global minimum is chosen from the lowest local minimums. The calculated temperatures after calibration match well with the real measurements.

Discussion. We remark here that it is much more difficult to assess the accuracy of the calculated on-chip temperatures, since we only have *one* on-chip diode readings but not the thermal distribution across the chip. Also, the accuracy of the thermal model is subject to the constraints from the environment such as room temperature changes, fan speed, aging of thermal interface material [33] etc. In such a senario, the scheduler should rely more on the thermal sensor readings as shown in Figure 3(b) to prevent error from accumulating. Nevertheless, our current model at least achieves the first order approximation to on-chip temperatures, and provides our thermal scheduler with reasonably good inputs.

7.2 Benchmark Classification

After model calibration, we ran 22 SPEC CPU2K benchmarks, mediabench, packetbench, and netbench, to first collect their temperature profiles and classify them into different thermal intensity groups.

For all the programs we ran, the IntReg is always among the hottest units. Since Pentium 4 has only one on-chip sensor to control the DTM, this sensor should be placed at a spot that is most likely to be the hottest. This spot is determined through extensive testing. To accommodate other hotspots, the threshold is lowered with enough headroom to account for the discrepancy between the temperature at the sensor and the real peak temperature at runtime. Overall, it is reasonable to assume that IntReg correctly represents the peak temperature at runtime.

Figure 5 shows the IntReg temperature profiles for all benchmarks executed back-to-back till completion. Here the starting temperature is ∼55°C, while that of the ambient air is ∼45°C, higher than the room temperature. As we can see, different programs present noticeably different thermal behavior: some run at a stable temperature, some have large variations, while others have sharp and spiky raises in temperature.

From the obtained thermal profiles, we can broadly classify the programs into three groups, *hot*, *warm*, and *cool*, according to their relative positions to each other. For example, `gcc` and `gap` produce the peak temperatures in Figure 5, and hence, are considered hot in the SPEC benchmarks. Similar principle is applied to the non-SPEC benchmarks as well. Note that if we combine the two groups of benchmarks, their relative temperature positions will change and the classification will be different. Our experiments separate these two groups of benchmarks due to their input sizes — SPECs have much larger inputs than the others, and they run significantly longer. The complete classification of these programs is shown in Table 1.

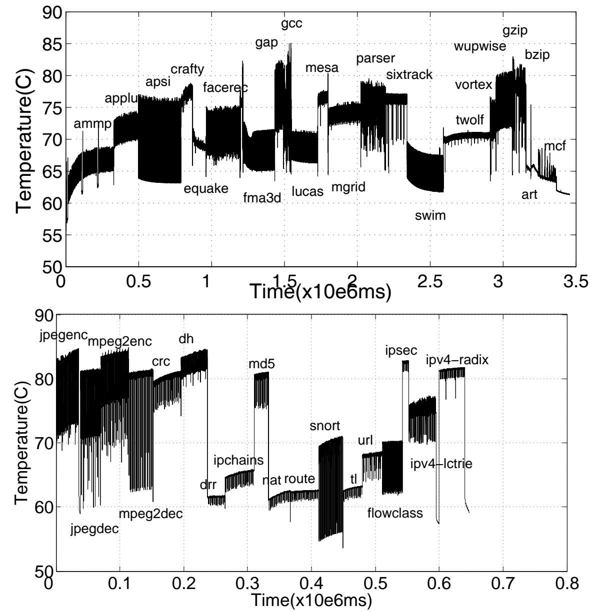


Figure 5. Thermal profiles of the IntReg for all 22 SPEC2K (above) and media, net, and packetbench (below).

SPEC 2K	
Hot	crafty gap gcc mesa mgrid sixtrack gzip bzip
Warm	applu apsi facerec parser vortex wupwise twolf
Cool	ammp equake fma3d lucas swim art mcf
Media, Packet bench, Netbench	
Hot	jpeg mpeg crc dh md5 ipsec ipv4_lctrie ipv4_radix
Warm	snort flowclass url ipchains
Cool	drr route tl nat

Table 1. Classifications of program thermal intensity.

7.3 Thermal Scheduling Results

We evaluate ThreshHot on different combinations of workloads, and compare the results with four other aforementioned scheduling algorithms. To avoid test space explosion, we limit the number of jobs executed simultaneously to 3. Every job can be hot, warm or cool, producing 10 combinations to test. The combinations where none of the jobs is hot are of little interest, since these will not involve thermal violations. Excluding those we are left with 6 combinations shown in Table 2.

We also want to consider the environmental conditions, in particular, the ambient temperature. The ambient temperature varies in response to activities in memory, disks or other components. This changes the temperature gradient, thus affecting

	SPEC2K	media, packet and netbench
HHH	mgrid gzip bzip	jpegdec ipv4_lctrie md5
HHW	gzip sixtrack vortex	jpegenc jpegdec flowclass
HHC	gzip bzip art	mpeg2enc mpeg2dec tl
HWW	gap apsi twolf	ipv4_lctrie url ipchains
HWC	gcc apsi art	ipv4_radix ipchains nat
HCC	mesa ammp mcf	dh drr route

Table 2. Workload combinations consisting of relatively hot (H), warm (W) and cool (C) jobs.

the efficiency of the heat removal. As a result, when the ambient temperature rises, cool programs can become warm, and warm programs can become hot to the CPU. Similarly, if the ambient temperature falls far below normal, even the hot programs, at their steady state, may not cause thermal violations.

To test the sensitivity of different schedulers to different environmental conditions, we varied the frequency scaling threshold from 75°C to 73°C and 71°C (from Figure 5, most programs' steady temperatures range between ∼60°C and ∼80°C). Such tests emulate, indirectly, the effect of varying the ambient condition, from low, through medium, to high, respectively. This is because our test environment had a steady temperature (26°C), setting for example, a low trip point such as 71°C results in relatively more DTMs, similar to a hot environment where more DTMs occur in the chip. This has been implemented through programming the OS clock modulation register to throttle the clock [43] upon reaching a pre-defined thermal threshold. Setting the threshold to even lower or higher values will not produce useful results, for it corresponds either to the case of all jobs being cool or all jobs being hot (which is the HHH case already tested in our study.)

7.3.1 DTM Reductions

Figure 6 shows the amount of DTMs for different workloads when executed under different schedulers. Each graph represents one thermal environment, as depicted by the labels. The results are normalized to the baseline DTM amount. Hence, the lower the bars, the better the results.

As we can see, in all workloads and in all thermal environments, the ThreshHot scheduler consistently removes more DTMs than other schedulers, often by a great amount. The reduction ranges are 8.4-81.9% (41.6% on average), 10.5-73.6% (34.5% on average), 2.5-48.5% (21.2% on average), and 4.1-70.5% (19.6%) for mild, medium, and harsh thermal environment, and non-SPEC benchmarks in medium environment respectively. The effectiveness of the ThreshHot over other schedulers is also evident. As an example, for workload 'HCC' in the medium thermal environment (Figure 6(b)), the MinTemp$^+$ scheduler reduced DTMs in the baseline schedule by 7.5%, the Random scheduler reduced 34.7%, while the ThreshHot scheduler reduced as high as 73.6%.

The Random scheduler performs slightly better than the MinTemp$^+$ scheduler. The former reduces more DTMs in mild and medium environments. However in harsh conditions, the Random scheduler can generate more DTMs than the base case, as shown in the 'HHW' workload in Figure 6(c) and (d). This is possible because among all workloads, this workload presents most DTMs in the baseline, as will be shown in Figure 7. The more DTMs in the baseline, the harder for the scheduler to remove them through randomizing the job order. Therefore, the Random scheduler may worsen the thermal behavior in a harsh thermal environment, or when most jobs are hot.

The Priority scheduler always increases the number of DTMs. For example, it increased the DTMs by 65% for the 'HCC' workload in the mild thermal environment (this cannot be seen from Figure 6(a) due to its scale). This is because the scheduler gives higher priorities (more CPU time) to the cool jobs than the hot jobs, so the former always finish sooner than the latter. As a result, the hot jobs, when cool jobs are exhausted, will trigger more DTMs than the baseline because the baseline always makes about the same progress for both jobs.

7.3.2 Performance Improvements

The performance improvements of different schedulers are not necessarily proportional to the amount of DTM reductions. This is because the time due to DTM is only a portion of the total execution time. Figure 7 plots the percentages of execution time attributed to DTMs. Figure 8 shows the overall performance improvements. The three subgraphs represent different thermal environments, similar to Figure 6. As expected, the ThreshHot scheduler consistently and significantly outperforms other schedulers. The Priority scheduler brings negative impact to performance unless there is constant supply of cool jobs, which was assumed in the original work [22]. From these graphs, we make the following observations:

- Workloads containing cool jobs incur fewer DTMs than those containing warm and hot jobs. Harsh thermal environment naturally causes more DTMs in all workloads.

- When considering Figure 6 and 7 jointly, we observe that the percentage DTM reduction rate depends on their contribution to the total execution time: the more execution time spent on DTMs, the less effective a thermal-aware scheduler is in removing them. (More precisely: it may remove more DTMs overall, but a smaller percentage.) For example, when the DTMs occur only 5.4% of time in 'HCC' (Figure 7(a)), the ThreshHot scheduler can easily remove 81.9% of them (Figure 6(a)). When the DTMs occur 44.4% of time in 'HHW' (Figure 7(c)), the TheshHot scheduler can only remove 2.5% of them (Figure 6(c)). Therefore, the amount of DTMs existing in a workload indicates directly how difficult it is to perform a thermal-aware scheduling. This is, of course, not surprising, for if the average temperature of the workload increases, so does the minimum number of DTMs in the optimal schedule – independently of what scheduler we use.

- Figure 8 shows the overall performance reduction, reflecting both the reduction of DTMs from Figure 6 and the original number of DTMs produced by the base scheduler, as seen in Figure 7. We see that a harsh/mild environment does not necessarily result in less/more performance improvements from a thermal-aware scheduler. Similarly, workloads having more cool jobs do not always result in most performance improvements. The highest performance improvements from the ThreshHot scheduler are seen in 'HHC' (6.56% in mild, 7.18% in medium, and 6.45% in harsh environment) and 'HCC' (6.31% in

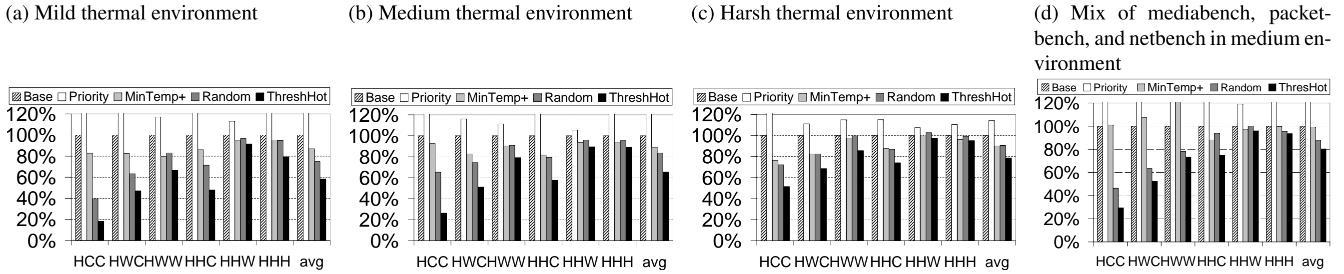


Figure 6. Number of thermal emergency triggers, normalized to the baseline scheduler (Linux default).

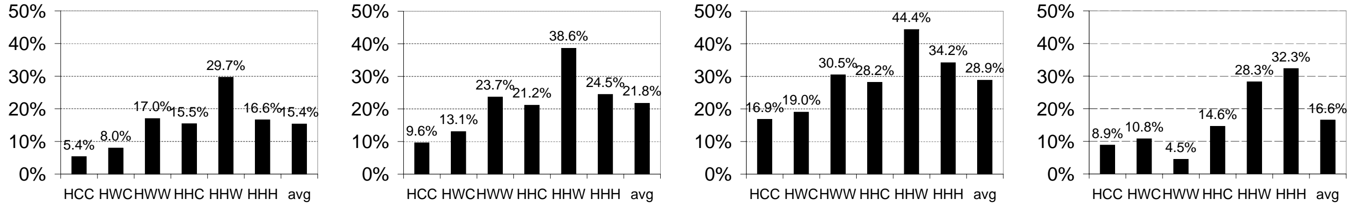


Figure 7. Percentage of execution time under DTM in the baseline scheduler.

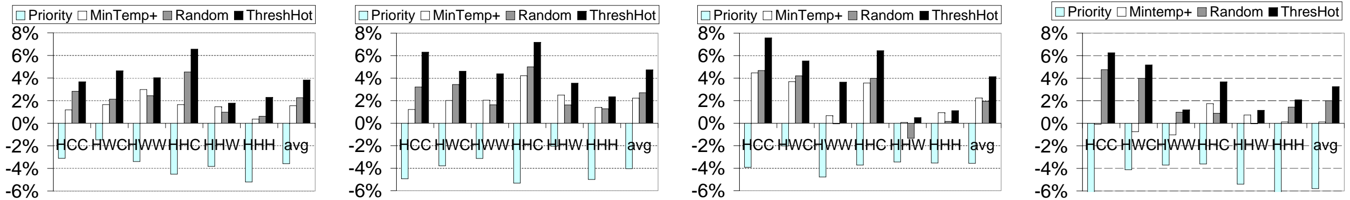


Figure 8. Performance improvements.

medium, 7.57% in harsh environment, and 6.25% in non-SPEC programs). The average improvements are 3.8%, 4.7%, 4.1%, and 3.25% for mild, medium, harsh thermal environment, and non-SPEC programs respectively.

We also observed that the MinTemp$^+$ scheduler, though far less effective than the ThreshHot scheduler, does a more consistent job in improving the total performance of a workload than the Random scheduler. The Random scheduler occasionally reduces the performance when it fails to remove DTMs, e.g., for 'HHW' in a harsh environment. However, when the conditions are mild or medium, the Random scheduler outperforms MinTemp$^+$, not only because it reduces more DTMs and has better performances, but also because it does not require any online power/temperature calculations, and thus is much easier to incorporate in an existing system. However, it tends to worsen the system performance when the thermal condition is severe.

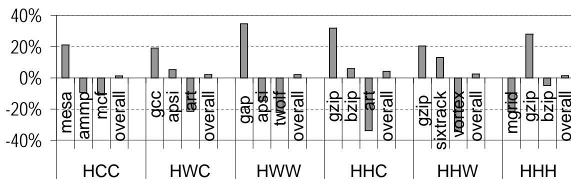


Figure 9. Drastic performance changes to individual jobs by MinTemp$^+$ scheduler (mild thermal environment).

One important downside of the MinTemp$^+$ scheduler is that it penalizes the cool slices for the thermal violations caused by hot slices. As we analyzed before, this is because the hot programs always run at full speed until the temperature increases above the threshold. Then the frequency is scaled and the coolest program is swapped in at the reduced frequency. As we can see from Figure 9, although the overall performance is improved in all workloads, each individual job experiences drastic performance changes, from ~-30% to ~+30%. In contrast, the performance gains from using the ThreshHot and the Random scheduler come mainly from the improvements in hot jobs, which is a more reasonable way of resolving the thermal emergencies.

7.3.3 Overhead

Finally, the overhead of our ThreshHot scheduler (and MinTemp$^+$ and Priority) mainly comes from the temperature calculation inserted in the kernel and context switches (including cache warm-up). We measured that the time required to calculate the temperatures is $\sim 16.45 \mu s$. This has been estimated by running the program with and without the temperature module in the kernel for a sufficiently long time and computing the temperature every $8ms$. This overhead includes probing the hardware performance counters, calculating power and calculating the temperatures using the method described in Section 4.1. As we also mentioned in Section 5.2, the average context switch time in our test system is $\sim 35.35 \mu s$. This has been determined by forcing periodic context switches among different programs, for different period lengths, and comparing the differences in execution time. The performance results presented earlier are based on real machine measurements and thus include all the overhead incurred at runtime.

8 Conclusion

We have proposed the ThreshHot scheduling algorithm, and compared it with three other thermal-aware schedulers. As we demonstrated, such a job scheduler, when carefully designed, not only is feasible but also can remove significant DTMs and

provide great performance benefits. Among the four schedulers we analyzed, our proposed ThreshHot scheduler follows the strategy of keeping the temperature right below but not exceeding a given threshold, based on the observation that this approach increases the heat removal rate and thus is likely to reduce the overall number of thermal violations. As it turns out, indeed, such a scheduling consistently removes most DTMs, and constantly improves the performance of all types of workloads in all thermal environments we tested.

The next scheduler we recommend is the Random scheduler, for it is easy to incorporate, and does not require any online power/thermal estimations. However, this scheduler does not achieve the same quality scheduling as the ThreshHot does, and tends to decrease the performance when the system is in a harsh thermal condition and DTMs happen very frequently.

The MinTemp$^+$ scheduler, since it toggles between the hot and cool job, is probably more suitable to lower the average temperature of a system. We emphasize that the threshold for toggling should be lower than the hardware threshold in order not to penalize the cool jobs unfairly. The Priority scheduler may be helpful in a system with long and cool jobs, and the hot jobs are not subject to any timing constraints.

References

[1] N. Bansal, T. Kimbrel, K. Pruhs, "Dynamic speed scaling to manage energy and temperature," *the 45th IEEE FOCS,* pp. 520-529, 2004.

[2] N. Bansal, K. Pruhs, "Speed scaling to manage temperature," *Symposium on Theoretical Aspects of Computer Science,* pp. 460-471, 2005.

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

[4] F. Bellosa, A. Weissel, M. Waitz, S. Kellner, "Event-driven energy accounting for dynamic thermal management," *Workshop on COLP,* 2003.

[5] D. Bovet, M. Cesati, "Understanding the Linux kernel, 3rd Edition," *O'Reilly Publisher,* November, 2005.

[6] D. Brooks, M. Martonosi, "Dynamic thermal management for high-performance microprocessors," *the 7th HPCA,* pp. 171-180, 2001.

[7] J. Choi, C.-Y. Cher, H. Franke, H. Hamann, A. Weger, P. Bose, "Thermal-aware task scheduling at the system software level," *ISLPED,* pp. 213-218, 2007.

[8] J. Choi, Y. Kim, A. Sivasubramaniam, J. Srebric, Q. Wang, J. Lee, "Modeling and managing thermal profiles of rack-mounted servers with thermostat," *IEEE 13th HPCA,* 2007.

[9] M.Chrobak and C.Dürr and M.Hurand and J.Robert, "Algorithms for temperature-aware task scheduling in microprocessor systems," submitted for publication.

[10] J. Donald, M. Martonosi, "Techniques for multicore thermal management: classification and new exploration," *the 33rd ISCA,* pp 78-88, 2006.

[11] S. H. Gunther, F. Binns, D. M. Carmean, J. C. Hall, "Managing the impact of increasing microprocessor power consumption," *Intel Technology Journal,* First Quarter, 2001.

[12] Y. Han, I. Koren, C. M. Krishna, "Temptor: A lightweight runtime temperature monitoring tool using performance counters," *Workshop on TACS,* 2006.

[13] L. He, W. Liao, M. R. Stan, "System level leakage reduction considering the interdependence of temperature and leakage," *DAC,* pp. 12-17, 2004.

[14] H. Hanson, S. Keckler, S. Ghiasi, K. Rajamani, F. Rawson, J. Rubio, "Thermal response to DVFS: Analysis with an Intel Pentium M," *ISLPED,* pp. 219-224, 2007.

[15] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, R. Bianchini, "Mercury and freon: temperature emulation and management for server systems," *ASPLOS,* pp. 106-116, 2006.

[16] S. Heo, K. Barr, K. Asanovic, "Reducing power density through activity migration," *ISLPED,* pp. 217-222, 2003.

[17] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, S. Velusamy, "Compact thermal modeling for temperature-aware design," *DAC,* pp. 878-883, 2004.

[18] W. Huang, E. Humenay, K. Skadron, M. R. Stan, "The need for a full-chip and package thermal model for thermally optimized IC designs," *ISLPED,* pp. 245-250, 2005.

[19] C. Isci, M. Martonosi, "Runtime power monitoring in high-end processors: methodology and empirical data," *MICRO,* pp. 93-104, 2003.

[20] R. Joseph, M. Martonosi, "Run-time power estimation in high-performance microprocessors," *ISLPED,* pp. 135-140, 2001.

[21] A. Krum, "Thermal management," In F. Kreith, editor *The CRC handbook of thermal engineering,* pp. 2.1-2.92. CRC Press, Boca Raton, FL 2000.

[22] A. Kumar, L. Shang, L.-S. Peh, N. Jha, "HybDTM: A coordinated hardware-software approach for dynamic thermal management," *DAC,* pp. 548-553, 2006.

[23] E. Kursun, C.-Y. Cher, A. Buyuktosunoglu, P. Bose, "Investigating the effects of task scheduling on thermal behavior," *Workshop on TACS,* 2006.

[24] Y. Li, B. Lee, D. Brooks, Z. Hu, K. Skadron, "CMP design space exploration subject to physical constraints" *HPCA,* Feb, 2006

[25] Y. Li, D. Brooks, Z. Hu, K. Skadron, "Performance, energy, and thermal considerations for SMT and CMP architectures," *HPCA,* pp. 71-82, 2005.

[26] Z. Lu, J. Lach, M. R. Stan, K. Skadron, "Improved thermal management with reliability banking," *IEEE Micro,* Nov. 2005.

[27] R. McGowen, "Adaptive designs for power and thermal optimization," *ICCAD,* pp. 118-121, 2005.

[28] P. C. Monferrer, G. Magklis, J. González, A. González, "Distributing the frontend for temperature reduction," *HPCA,* pp. 61-70, 2005.

[29] J. Moore, J. Chase, P. Ranganathan, R. Sharma, "Making scheduling 'cool': temperature-aware workload placement in data centers," *USENIX 2005 Annual Technical Conference,* pp. 61-75, 2005.

[30] P. Pillai, K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," *ACM SOSP,* pp. 89-102, 2001.

[31] M. D. Powell, M. Gomaa, T. N. Vijaykumar "Heat-and-run: leveraging SMT and CMP to manage power density through the operating system," *ASPLOS,* pp. 260-270, 2004.

[32] E. Rohou, Michael D. Smith, "Dynamically managing processor temperature and power," *Workshop on FDDO,* 1999.

[33] E. Samson, S. Machiroutu, J-Y. Chang, I. Santos, J. Hermerding, A. Dani, R. Prasher, D. W. Song, "Interface material selection and a thermal management technique in second-generation platforms built on Intel Centrino mobile technology," *Intel Technology Journal,* February 2005.

[34] K. Skadron, T. Abdelzaher, M. R. Stan, "Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management," *HPCA,* pp. 17-28, 2002.

[35] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, D. Tarjan, "Temperature-aware microarchitecture," *ISCA,* pp. 2-13, 2003.

[36] J. Srinivasan, S. V. Adve, "Predictive dynamic thermal management for multimedia applications," *ICS,* pp. 109-120, 2003.

[37] J. Stoer, R. Bulirsch, "Introduction to numerical analysis," *Springer-Verlag,* 2nd ed. 1991.

[38] B. Sprunt, "Brink and abyss Pentium 4 performance counter tools for Linux," *TR,* February 2002.

[39] S. Wang, R. Bettati, "Reactive speed control in temperature-constrained real-time systems," *the 18th Euromicro Conference on Real-Time Systems,* 2006.

[40] S. Wang, R. Bettati, "Delay analysis in temperature-constrained hard real-time systems with general task arrivals," *RTSS,* pp. 323-334, 2006.

[41] W. Yuan, K. Nahrstedt, "Energy-efficient soft real-time CPU scheduling for mobile multimedia systems," *ACM SOSP,* pp. 149-163, 2003.

[42] N. Zeldovich, R. Chandra, "Interactive performance measurement with VNCplay," *FREENIX Track: USENIX Annual Technical Conference,* 2005.

[43] "Intel Pentium 4 processor in the 478-pin package thermal design guidelines," design guide, Intel, May 2002.

[44] *Predictive Technology Model,* http://www.eas.asu.edu/ ptm/