Performance-Aware Thermal Management via Task Scheduling

XIUYI ZHOU and JUN YANG
University of Pittsburgh
MAREK CHROBAK
University of California, Riverside
and
YOUTAO ZHANG
University of Pittsburgh

High on-chip temperature impairs the processor's reliability and reduces its lifetime. Hardware-level dynamic thermal management (DTM) techniques can effectively constrain the chip temperature, but degrades the performance. We propose an OS-level technique that performs thermal-aware job scheduling to reduce DTMs. The algorithm is based on the observation that hot and cool jobs executed in a different order can make a difference in resulting temperature. Real-system implementation in Linux shows that our scheduler can remove 10.5% to 73.6% of the hardware DTMs in a medium thermal environment. The CPU throughput is improved by up to 7.6% (4.1%, on average) in a severe thermal environment.

Categories and Subject Descriptors: B.8 [**Hardware**]: Performance and Reliability; D.4.1 [**Operating Systems**]: Process Management

General Terms: Algorithms, Management, Performance

Additional Key Words and Phrases: Thermal management, task scheduling

ACM Reference Format:

Zhou, X., Yang, J., Chrobak, M., and Zhang, Y. 2010. Performance-aware thermal management via task scheduling. ACM Trans. Architec. Code Optim. 7, 1, Article 5 (April 2010), 31 pages. DOI = 10.1145/1746065.1736070 http://doi.acm.org/10.1145/1746065.1736070.

5

This is an Extension of Conference Paper of "*Dynamic Thermal Management through Task Scheduling*. ISPASS 2008: 191–201." An itemized list of the additional material is provided separately to explain how 25% new material is added.

Author's address: X. Zhou, Department of Electrical & Computer Engineering, University of Pittsburgh, Pittsburgh, PA 15261; email: xiz44@pitt.edu; J. Yang, Department of Electrical & Computer Engineering, University of Pittsburgh, Pittsburgh, PA 15261; email: junyang@ece.pitt.edu; M. Chrobak, Department of Computer Science & Engineering, University of California, Riverside, CA 92521; email: marek@cs.ucr.edu; Y. Zhang, Department of Computer Science & Engineering, University of Pittsburgh, Pittsburgh, PA 15261; email: zhangyt@cs.pitt.edu.

DOI 10.1145/1746065.1736070 http://doi.acm.org/10.1145/1746065.1736070

1. INTRODUCTION

As technology for microprocessors enters the nanometer regime, power density has become one of the major constraints on 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 [Intel 2002], are designed for typical, not worst-case, thermal conditions. In modern chip architectures, these challenges are addressed by implementing appropriate dynamic thermal management (DTM) modules that regulate chip temperature at runtime.

There have been numerous studies on DTMs at the microarchitecture level [Brooks and Martonosi 2001; Gunther et al. 2001; Heo et al. 2003; Li et al. 2005; Monferrer et al. 2005; Skadron et al. 2002; Skadron et al. 2003; Srinivasan and Adve 2003]. Architecture solutions can respond to thermal crises 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 [Choi et al. 2007; Donald and Martonosi 2006; Hanson et al. 2007; Kumar et al. 2006; Kursun et al. 2006; Powell et al. 2004]. The main idea is to exploit variations of chip's temperature resulting from executing jobs with different CPU usage profiles. Some "hot" jobs, such as algorithms involving intensive computation, cause the chip to run at a higher temperature than "cool" jobs, where most work involves data transfers between memory. By swapping such hot and cool jobs at an appropriate time, we can control the chip's temperature. This approach has been proposed for both CMPs [Choi et al. 2007; Donald and Martonosi 2006; Powell et al. 2004] and single-core processors [Hanson et al. 2007; Kumar et al. 2006; Kursun et al. 2006]. 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–hot 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 [Kumar et al. 2006]. To know which job will be hot or cool for the hotspot, we develop a highly efficient online temperature estimator, leveraging the performance counter-based power estimation [Isci and Martonosi 2003; Joseph and Martonosi 2001; Kumar et al. 2006], compact thermal modeling [Skadron et al. 2003], and a fast temperature solver [Han et al. 2006]. 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 article is organized as follows: Section 2 discusses previous related works. Section 3 elaborates on our thermal-aware heuristic algorithm and justifies its fundamental principle 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 article.

2. PRIOR WORK

Some recent works have developed temperature-control techniques for regular [Rohou and Smith 1999] and real-time [Bansal et al. 2004; Bansal and Pruhs 2005; Wang and Bettati 2006b; Wang and Bettati 2006a; Hanumaiah et al. 2009] 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 or as early as possible. Similar approaches can be used to minimize energy consumption in real-time systems [Pillai and Shin 2001; Yuan and Nahrstedt 2003]. Further, temperature control can be achieved through adjusting microarchitectural parameters, such as instruction window size and issue width, which have relatively lower overhead than DVFS-based algorithms [Jayaseelan and Mitra 2008; Jung et al. 2008; Khan and Kundu 2008]. Temperature control through job scheduling has also been utilized to enhance the reliability of a processor [Lu et al. 2005; Coskun et al. 2008; Coskun et al. 2008]. 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

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

assignment and migration to *balance* the chip temperature at runtime [Powell et al. 2004]. In another work [Donald and Martonosi 2006], a suite of DTM techniques, job migration policies, and control granularity are jointly investigated to achieve the maximum chip throughput. Also, a simple periodic thread swapping between two cores to balance the chip temperature was studied on a dual-core processor [Choi et al. 2007]. Recently, an accurate temperature prediction model was developed to make job scheduling more precise [Yeo et al. 2008]. Moreover, due to the increasing on-die parameter variations in deep submicron dimensions, job scheduling is enhanced with variation sensing to assist the system power/thermal management to effectively manage the existing on-chip variation [Kursun and Cher 2008]. 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. Our preliminary study [Yang et al. 2008] showed that such a technique is better than simple hot–cool job interleaving. In this article, we demonstrate this philosophy using a scheduling heuristic on a single-core processor and leave its extensions to CMPs as future work. Compared with Yang et al. [2008], this article adds a fifth Greedy algorithm for comparison. Furthermore, the article analyzes the impact of power misprediction and gives a quantitative measurement on the performance upper bound a thermal-aware scheduler can achieve. Our analyses show that the gain of any more complex power prediction algorithm is marginal. Finally, we also conducted analyses and experiments on the scalability of our algorithm to see the scheduling overhead would grow linearly with the number of jobs. Our results show that our proposed algorithm is scalable.

In single-core domain, the "HybDTM" [Kumar et al. 2006] 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 Bellosa et al. [2003], 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 Moore et al. [2005], a temperature-aware workload placement heuristic was studied for data centers to minimize the cost of cooling. The "mercury and freon" [Heath et al. 2006] framework uses a software to estimate temperatures for a server cluster and manages its component temperatures through a thermal-aware load balancer. The "ThermoStat" [Choi et al. 2007] 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-AWARE SCHEDULING ALGORITHMS

When the processor is 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, and so on. Thus, independent of the characteristics and focus of a given system, processor overheating will negatively affect its performance.

The frequency of thermal violations clearly depends on the workload, since variations in CPU intensity between different jobs result in variations in the amount of heat generated during their execution. We cannot control the processor workload, but we can control the order in which the jobs in this workload are executed. The objective of this research was to show that significant performance improvements can be achieved by thermal-aware scheduling, namely using the information about the thermal behavior of the system to schedule the jobs in a given workload so as to minimize the number of thermal violations.

To further delineate the difference between our approach and most of the previous work, we emphasize that we do not attempt to minimize the temperature itself; we do control the temperature, but not for its own sake. The philosophy behind our approach is that, since the temperature is controlled by a hardware DTM system, we do not need to be concerned with the issues of safety and reliability—the DTM ensures that the chip runs at a safe temperature, and the variations of the temperature below the thermal threshold have only negligible, if any, impact on the reliability and expected lifespan of a chip. With the DTM in place, the only noticeable impact of high temperature is on chip's performance; thus, we focus on minimizing this impact.

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. In our scheduling framework, we incorporate thermal awareness into this job selection strategy while keeping all other features intact.

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 interval $t+1$ is higher than for the order yx (y before x). This means that, as long as the temperatures for both orders stay below the threshold, the 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 [Krum 2000] provides a convenient basis for modeling the chip temperature using a dynamic compact thermal model [Skadron et al. 2003]:

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

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:

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

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

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

while for schedule yx, this final temperature will be

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

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 beneficial 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

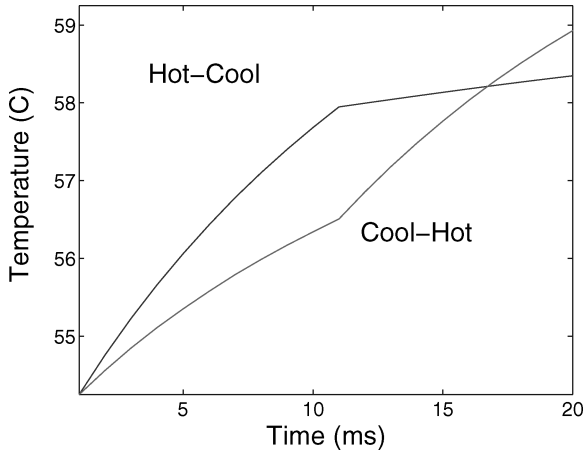

Fig. 1. The impact of scheduling a hot and cool program in different orders.

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.

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 given earlier in the text, 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 previously mentioned 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 [Chrobak et al. 2008]); 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 $\mathcal{P}$, that is, 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 (this follows essentially the same argument as the one described previously).

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

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 earlier, 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:

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

where the first four extra terms describe the heat transfer from the central node (with temperature T) to its lateral neighbor nodes (with temperatures $T_1 - T_4$) and R_{Li} denoting its lateral resistance from the central node to the i-th neighbor. Equation (6) describes the model where each node has four neighbors. In general, the number of neighbors per node depends on the processor floorplan and on how the system is discretized, and Equation (6) can be easily adapted to those other models.

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 [Donald and Martonosi 2006; Powell et al. 2004; Skadron et al. 2003]. In more quantitative terms, our experience with a full-chip model shows that the R_{Li}'s are typically 10 to 20 times the R for a hot unit such as the IntReg. The resulting lateral RC time constants are on the order of 100ms and vertical RC time constant is less than 10ms. 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 previously, 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 [Intel 2002].

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 conductance. 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 predicted from its past power consumption), and Δt (which is a fixed value).

Temperature calculation. It may seem that solving Equation (7) requires a lot of computation for the scheduler, possibly creating excessive overhead when performed 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 [Han et al. 2006], which 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. Equation (8) simplifies now to:

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

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 Equation (9) is time invariant,

it holds for every interval Δt. Therefore:

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

As we can see, once $\mathbf{A}$ and $\mathbf{B}$ are precalculated 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 [McGowen 2005]. 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 [Bellosa 2000; Isci and Martonosi 2003; Joseph and Martonosi 2001]. 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 FU's 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 [Huang et al. 2005]. 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 He et al. [2004] and Li et al. [2006] using PTM 0.13μ technology parameters [NIMC 2007], 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 trade-off 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 $\sim$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

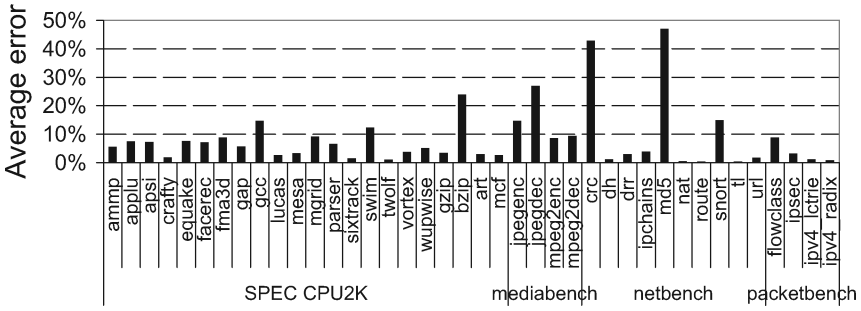

Fig. 2. Average error rates for the last power value predictor.

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 misscheduling. This is easy to explain: In order for a major misscheduling 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 misscheduling 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.

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).

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 there are very few sensors, probing the counters is still necessary, but the sensor readings can be used for online self-calibration to lessen the error due to thermal and power model abstraction.

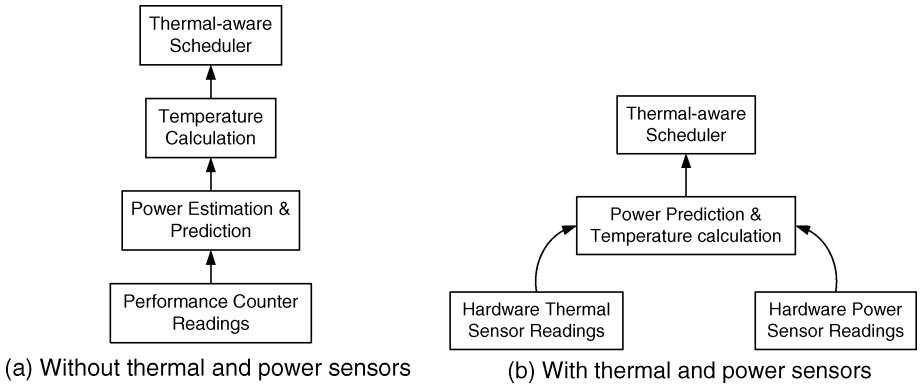

Fig. 3. Thermal-aware task scheduling methodologies.

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 [Bovet and Cesati 1984] 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, for example, 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 to 10ms. 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 [Donald and Martonosi 2006; Powell et al. 2004]. 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 [Sprunt 2002]), 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 four (or eight) 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 [Bovet and Cesati 1984]. 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 0.44%. 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. A slice may be delayed (with respect to its execution time in the standard Linux scheduler), either because warmer eligible slices are available, or because it is too hot for execution, but it will never be postponed beyond the current epoch. Essentially, our policy can be thought of as rearranging job slices within an epoch, while preserving the epochs themselves. Thus, viewed on the epoch scale, all jobs are processed at exactly at the same rate as in the standard Linux scheduler.

Impact on nonbatch 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 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 and Chandra [2005], 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) GREEDY. This algorithm always first selects the coolest job to run in an epoch, and postpones hot job intervals as late as possible. This is an intuitive and simple modification of the default Linux scheduler.
(2) RANDOM. This algorithm randomly selects a job to execute in every scheduling interval (8ms). As a result, it intersperses the slices of the jobs in the current batch. The produced schedule has the property that, in expectation, at each time of the scheduling interval the heat generation rate is the same. We test this scheduler to measure whether the performance improvements can be attributed simply to frequent context switches and to distributing the total heat contribution of the batch equally over the scheduling interval. By showing that our proposed algorithm outperforms RANDOM, we can argue that a guided job selection is needed for controlling the temperature.
(3) PRIORITY. This algorithm lowers the priority of the hot jobs and raises the priority of the cool jobs for every new epoch [Kumar et al. 2006]. A job is considered "hot" if it's overall temperature in an epoch exceeds a predefined 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.
(4) MINTEMP$^+$. This algorithm selects the coolest job if the current chip temperature is over the threshold and selects the hottest job if the current temperature is below the threshold. We improved the original design of MINTEMP [Kursun et al. 2006] 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: (i) there are often great temperature variations within jobs (Figure 5 shows this property) and (ii) 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, it helps the scheduler follow the policy correctly.
(5) 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 five schedulers. For clarity, two epochs are shown and all graphs have the same baseline scheduling results so that the differences among the five 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

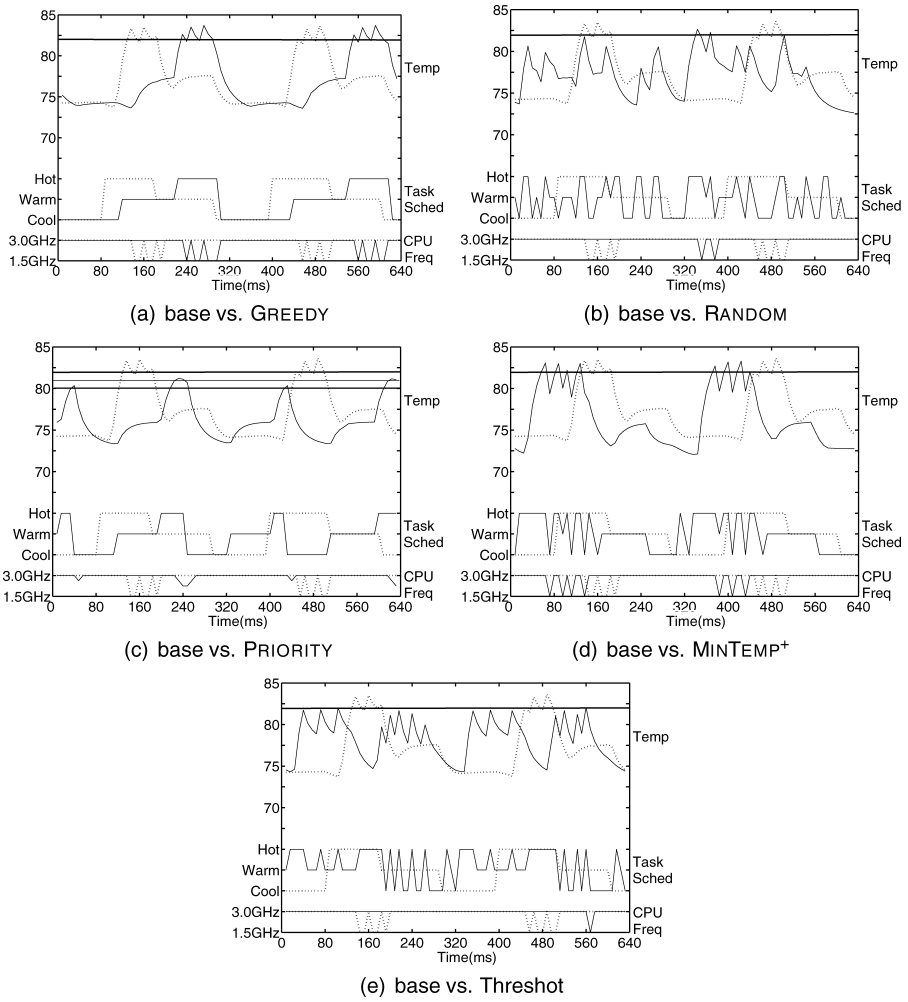

Fig. 4. A close-up of the execution traces for five different algorithms. Each graph compares the default Linux scheduler (dashed line) with one algorithm (solid line). In all graphs, the top portion shows the temperature variation with time. The middle portion shows the job switching sequence, and the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

threshold three times per epoch. This can 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 awhile. We now compare and contrast how the other five schedulers impact the peak temperatures.

GREEDY *scheduler.* The GREEDY scheduler always picks the coolest job interval to run. As shown in Figure 4(a), the jobs are executed in ascending temperature order. Thus, GREEDY tends to postpone the execution of hot intervals toward the end of the epoch. At that time, thermal violations cannot be avoided as only hot

job intervals are pending for execution. We can see from the figure that the hot job also triggers three frequency scalings by the end of each epoch.

RANDOM *scheduler.* As shown in Figure 4(b), the RANDOM scheduler switches to a different job, randomly picked from the job pool, on every scheduling interval. 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, for example, in the first epoch, but cannot prevent the scalings 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 shown in from Figure 4(c), 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 discussed later in the text, 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(d). As we can see, at the beginning of an epoch when the 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 shown 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(e), 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.

7. EXPERIMENTAL EVALUATION

In our thermal-aware scheduling, the performance is improved through temperature control. 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 [Huang et al. 2004; Huang et al. 2005; Skadron et al. 2003] 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 $\mathbf{A}$ and $\mathbf{B}$. All these values depend on the processor- and package-dependent thermal RC. From our experience, 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: (i) an on-chip thermal diode reading, (ii) a thermometer reading on the heat spreader, (iii) a thermometer reading on the heat sink, and (iv) 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:

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

where the x_is are our parameters to be adjusted. This is a minimization problem that can be tackled by the *conjugate gradient* (CG) method [Stoer and Bulirsch 1991], 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 value 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 [Samson et al. 2005], and so on. In such a scenario, 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 $\sim$55°C, while that of the ambient air is $\sim$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.

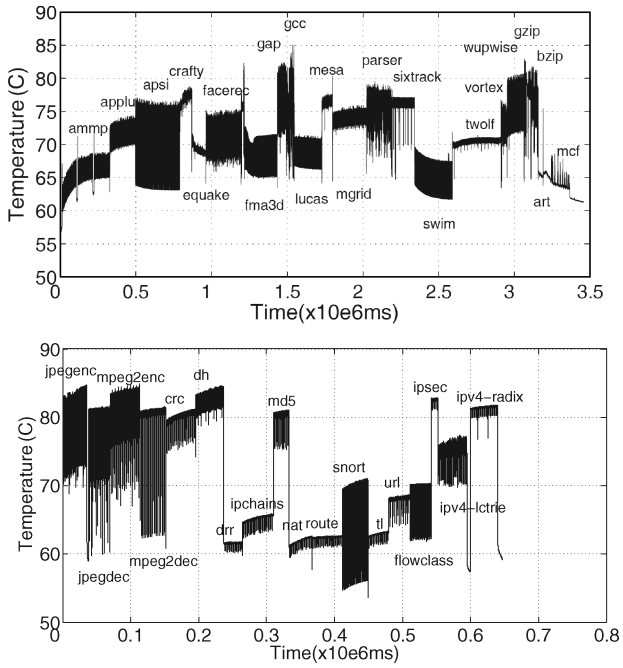

Fig. 5. Thermal profiles of the IntReg for all 22 SPEC2K (top) and media, net, and packetbench (bottom).

Table I. Classifications of Program Thermal Intensity

SPEC 2K	
Hot	crafty gap gcc mesa sixtrack gzip bzip vortex
Warm	applu apsi facerec mgrid parser 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

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 I.

Table II. Workload Combinations Consisting of Relatively Hot (H), Warm (W), and Cool (C) Jobs

	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

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 three. 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 the six combinations shown in Table II.

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 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). With a steady test-environment temperature (26°C in our case), lowering the threshold to, say, 71°C results in relatively more DTMs than for higher thresholds, quite similar to retaining the threshold while running the same workload with a higher ambient temperature. Therefore, such tests emulate, indirectly, the effect of varying the ambient condition, from low, through medium, to high, respectively. These tests have been implemented through programming the OS clock modulation register to throttle the clock [Intel 2002] upon reaching a predefined 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. We do not show the results for the Greedy scheduler, since it is almost always worse than the baseline scheduler.

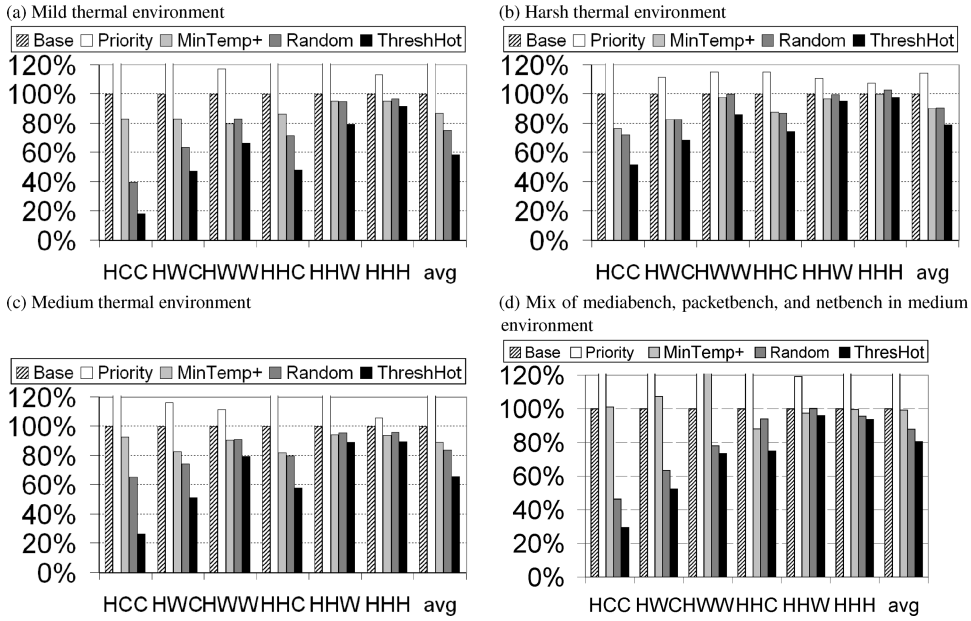

Fig. 6. Number of thermal emergency triggers, normalized to the baseline scheduler (Linux default).

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 to 81.9% (41.6%, on average), 10.5 to 73.6% (34.5%, on average), 2.5 to 48.5% (21.2%, on average), and 4.1 to 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(c)), 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 Figures 6(b) and 6(d). This is, by itself, an interesting phenomenon, and can be explained as follows. What RANDOM does, is, in essence, to replace the batch by one long job whose temperature (or heat contribution rate) is the "average" of those of the jobs in the batch. For mild and medium environments, this average value is below the threshold, and as a result, the RANDOM's schedule stays below the threshold for most of the time, reducing the number of thermal violations. But if this average is above the threshold, like in the "HHW" workload, the thermal violations will occur throughout the whole interval. In contrast, in the base schedule, they occur on the hot jobs but not on the warm job. Therefore, in

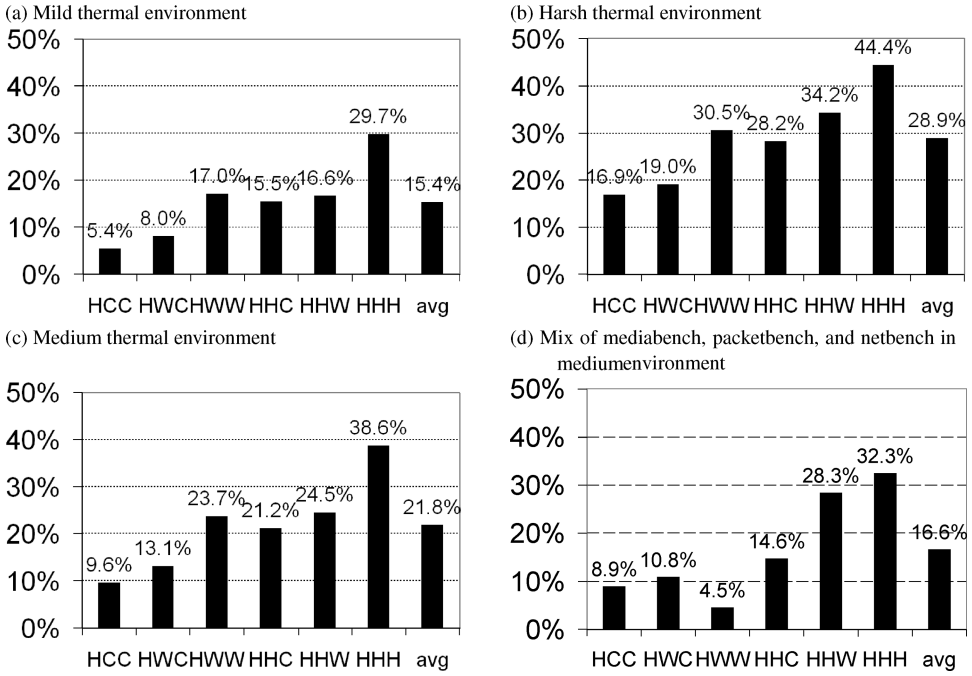

Fig. 7. Percentage of execution time under DTM in the baseline scheduler.

this case, RANDOM will actually create more threshold violations than the base scheduler.

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 is not shown in 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 a constant supply of cool jobs, which was assumed in the original work [Kumar et al. 2006]. 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.

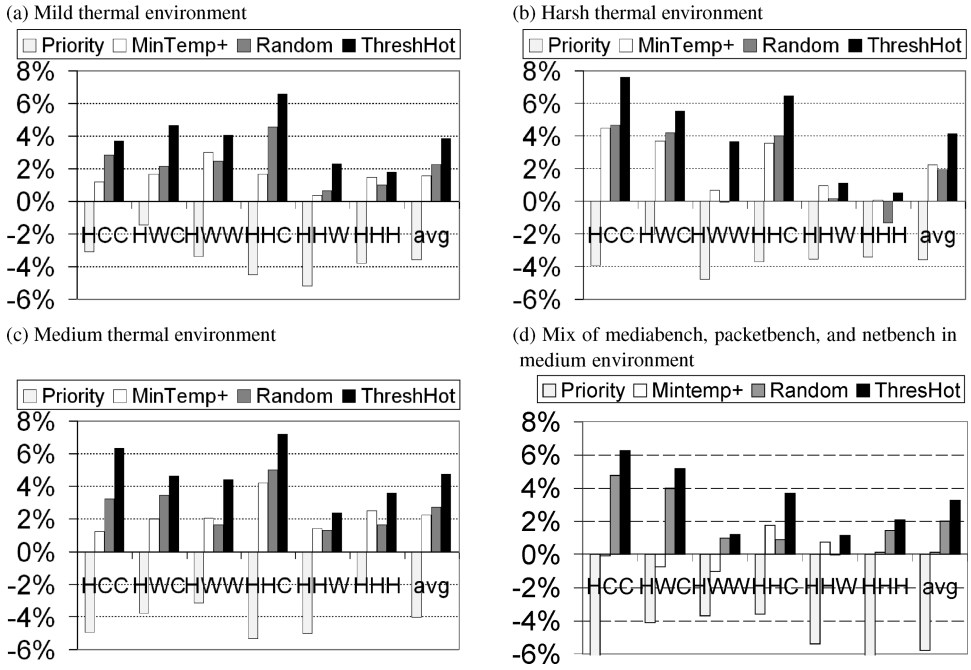

Fig. 8. Performance improvements.

—When considering Figures 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 ThreshHot 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 improvement, 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 medium, 7.57% in harsh environment, and 6.25% in non-SPEC programs). The average improvements are 3.8%, 4.7%, 4.1%,

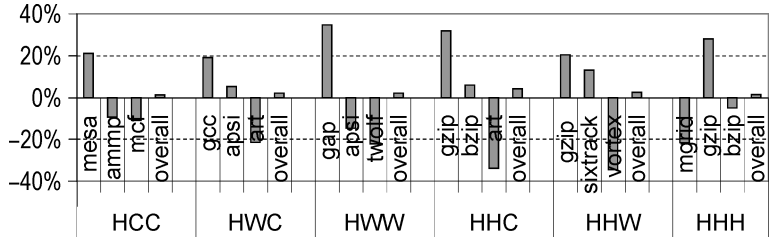

Fig. 9. Drastic performance changes to individual jobs by MINTEMP+ scheduler (mild thermal environment).

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, for example, 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; thus, it is much easier to incorporate in an existing system. However, it tends to worsen the system performance when the thermal condition is severe.

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 shown in Figure 9, although the overall performance is improved in all workloads, each individual job experiences drastic performance changes, from $\sim -30\%$ to $\sim +30\%$. 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 *Impact of Power Misprediction.* Our proposed THRESHHOT scheduler relies on the projected temperatures to make a selection for the next scheduling interval. As we discussed in Section 4.2, the temperature in the next interval will depend on a job's power consumption in the next interval, which is predicted from the current interval. Figure 2 shows the percentages of errors in predicted power values using the last-value prediction. In this section, we quantify the impact of such errors in performance improvement.

Our goal is to compare the last-value power predictor with an oracle power predictor and see their contributions on performance improvement under the THRESHHOT scheduler. To achieve this, we collected the power traces from the baseline scheduler and perform the THRESHHOT scheduling twice offline, one with the last-value power predictor and another with the oracle predictor. In our scheduler, the power predictor works with the scheduler in the following way.

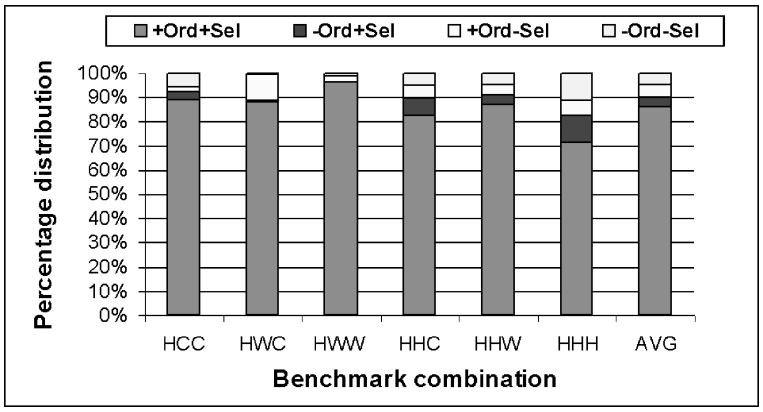

Fig. 10. The distribution of last-value prediction results.

First, the predicted powers are used to calculate the temperature rises in the next step. Second, the temperatures are sorted from high to low. Third, search temperatures from high to low and find the first one below the threshold. As we can see, if the last-value predictor and oracle predictor come up with the same temperature order and selected the same job to run, then the two predictors are equally good. Also, even when the temperature orders are different, if the two predictors happen to select the same job to run, they are still equally good. For example, the last-value predictor may generate a job temperature order from high to low as: 2, 1, 3, and the threshold is between 2 and 1, so job 1 should be selected. The oracle predictor, on the other hand, generates an order as 1, 2, 3, and 1 is below the threshold. Therefore, even when the last-value predictor made a mistake, as long as the right job is selected, the scheduling decision is still correct.

Figure 10 shows the percentage distribution of four possibilities of the last-value power prediction results, from bottom up: correct temperature order and correct job selection ("+Ord+Sel"), incorrect temperature order but correct job selection ("-Ord+Sel"), correct temperature order but incorrect job selection ("+Ord-Sel"), incorrect temperature order and incorrect job selection ("-Ord-Sel"). On average, the last-value predictor can result in 85.72% of "+Ord+Sel", and 4.44% of "-Ord+Sel", totaling 90.16% of correct scheduling decision. This is fairly significant considering the simplicity of the predictor. Figure 11 further shows the performance speed-ups for the last-value predictor and the oracle power predictor. As we can see that, on average, the last-value predictor achieves only 0.6% less speed-up than the oracle power. Therefore, we conclude that designing complex power prediction schemes may not pay off, since the additional performance improvement will be marginal.

7.3.4 *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$16.45μs. This has been estimated by running the program with and without the temperature

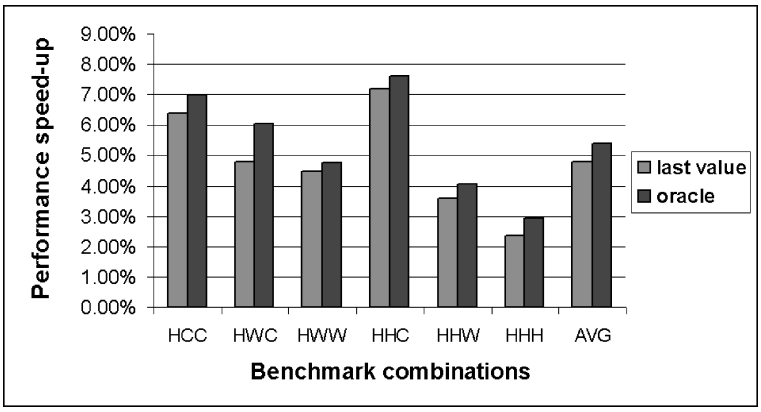

Fig. 11. The offline performance comparison of last-value power predictor and oracle power predictor.

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 mentioned in Section 5.2, the average context switch time in our test system is $\sim$35.35μs. 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.

The most concern of the scheduling overhead is whether the algorithm can be scaled up to support more jobs. With more number of jobs, more time is required to calculate the next step temperatures and make a scheduling decision. Note that the context switch overhead remains the same because there is still only one switch no matter how many candidates there are. Therefore, we only need to limit the time spent in calculating temperatures for all jobs. This can be achieved using the following optimization. First, sort the next-step powers for all jobs from high to low. The next-step temperature results corresponding to those powers will be monotonically decreasing. To avoid calculating temperatures for all powers, we can do a binary search to find the highest power that generates a temperature below the threshold. This can reduce the number of temperature calculations from N to $O(\log N)$, where N is the number of jobs. Such an optimization provides a scalable solution to our algorithm.

To verify the scalability of our algorithm, we measured the scheduling overhead when the number of jobs increases. The scheduling overhead includes time for both temperature calculations and context switches. When we increase the number of jobs, the performance penalty due to DVFS varies due to the changing relative thermal intensity in the job mix. Therefore, we suppress the engagement of all DVFS to remove the noise in the scheduling overhead. We measured the overhead for both RANDOM and THRESHHOT, and compared them with the baseline. The results are shown in Figure 12.

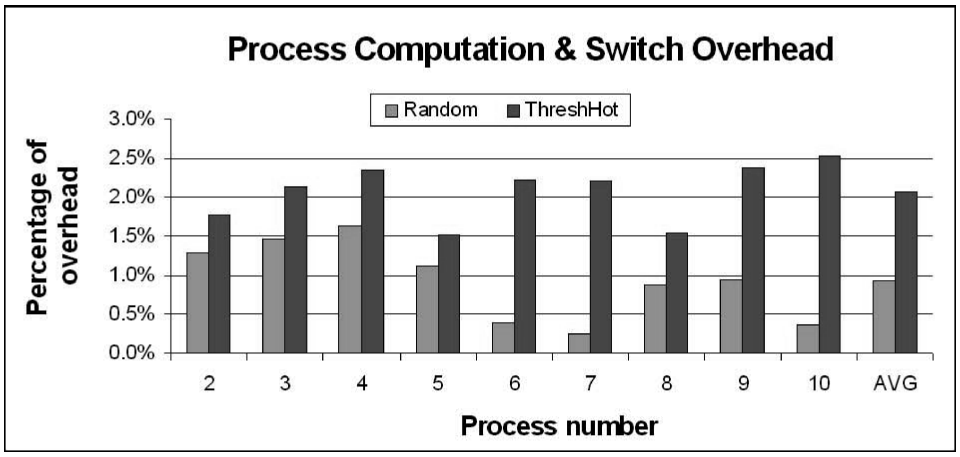

Fig. 12. The overhead from context switch and temperature computation.

The overhead is calculated as the percentage of extra execution time required by RANDOM and THRESHHOT, compared to the baseline. The RANDOM scheduler incurs mainly the context switch overhead, as it does not need to project the temperature variation of the jobs, and randomly picks one to execute in the next time window. Hence, its overhead is relatively constant irrespective of the number of jobs. The results show that the average overhead is 0.93%, with a maximum of 1.64% for scheduling four jobs and a minimum of 0.25% for seven jobs. These results confirm that frequent context switches incur insignificant overhead to the overall performance. The THRESHHOT scheduler shows additional overhead in temperature calculations for all job mixes. As we explained earlier, the temperature calculations are necessary for only $\log N$ jobs. We conservatively assumed there can be up to 10 active jobs for scheduling on a single core. In reality, this number is likely to be much smaller. Thus, the temperature calculation is performed between one and four jobs. The actual time depends on specific temperature values of different jobs. That is, more number of jobs does not necessarily incur more temperature calculation time. As we can see from the results, there is no clear trend in increasing overhead from 2 to 10 jobs. On average, we see a 2.07% performance overhead including both temperature calculation and context switch. The highest overhead of 2.52% is seen in scheduling 10 jobs, and the lowest of 1.51% is seen for scheduling 5 jobs. Our early results in Section 7.3 were for scheduling three jobs. As we can see here, the scheduling overhead for three jobs is around the average. Therefore, we conclude that our proposed THRESHHOT is a scalable solution.

Looking into future CMPs where number of jobs increases propotionally with the number of cores, the problem becomes how to assign jobs to balance thermal behaviors among cores. For example, for CMPs with 64 cores and 300+ jobs, each core will be assigned around five jobs in its local job queue. The question is how to select 5 jobs from a pool of 300+ jobs. One possible solution is to sort the jobs according to their power history and then assign each core with equal number of hot and cool jobs. Such sorting and subsequent job migration could

be done once every several seconds, to keep the overhead as small as possible. After the job assignment is done, each core will still only incur 1.5% overhead if five jobs are assigned to the core. If using our ThreshHot algorithm, each core can gain ∼5% average performance improvement, still achieving positive gains while regulating the chip's thermal behavior.

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 can also remove a significant fraction of DTMs and provide great performance benefits. Among the four schedulers we analyzed (with the experimental results of the GREEDY scheduler omitted), 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, such a scheduling consistently removes most DTMs and 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 workload with long and cool jobs and where the hot jobs are not subject to any timing constraints.

REFERENCES

BANSAL, N., KIMBREL, T., AND PRUHS, K. 2004. Dynamic speed scaling to manage energy and temperature. In *Proceedings of the 45th Annual Symposium on Foundations of Computer Science.* IEEE, Los Alamitos, CA, 520–529.

BANSAL, N. AND PRUHS, K. 2005. Speed scaling to manage temperature. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*. Springer, Berlin, 460–471.

BELLOSA, F. 2000. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of the 9th European Workshop*. ACM, New York.

BELLOSA, F.,WEISSEL, A.,WAITZ, M., AND KELLNER, S. 2003. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low-Power.* IEEE, Los Alamitos, CA.

BOVET, D. AND CESATI, M. 1984. *Understanding the Linux Kernel* 3rd Ed. O'Reilly, Sebastopol, CA.

BROOKS, D. AND MARTONOSI, M. 2001. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture.* IEEE, Los Alamitos, CA, 171–180.

CHOI, J., CHER, C.-Y., FRANKE, H., HAMANN, H., AND BOSE, A. W. P. 2007. Thermal-aware task scheduling at the system software level. In *Proceedings of International Symposium on Low-Power Electronics and Design*. ACM, New York, 213–218.

CHOI, J., KIM, Y., SIVASUBRAMANIAM, A., SREBRIC, J., WANG, Q., AND LEE, J. 2007. Modeling and managing thermal profiles of rack-mounted servers with thermostat. In *Proceedings of 13th International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA.

CHROBAK, M., DURR, C., HURAND, M., AND ROBERT, J. 2008. Algorithms for temperature-aware task scheduling in microprocessor systems. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*. Springer, Berlin, 120–130.

COSKUN, A., ROSING, T., AND GROSS, K. 2008. Proactive temperature balancing for low- cost thermal management in mpsocs. In *Proceedings of the International Conference on Computer-Aided Design*. IEEE, Los Alamitos, CA, 250–257.

COSKUN, A., ROSING, T., WHISNANT, K., AND GROSS, K. 2008. Static and dynamic temperature-aware scheduling for multiprocessor socs. *IEEE Trans. VLSI Syst. 16*, 9, 1127–1140.

DONALD, J. AND MARTONOSI, M. 2006. Techniques for multicore thermal management: Classification and new exploration. In *Proceedings of the 33rd International Symposium on Computer Architecture*. ACM, New York, 78–88.

GUNTHER, S. H., BINNS, F., CARMEAN, D. M., AND HALL, J. C. 2001. Managing the impact of increasing microprocessor power consumption. *Intel Tech. J.* ftp://download.intel.com/technology/itj/q12001/pdf/art_4.pdf

HAN, Y., KOREN, I., AND KRISHNA, C. M. 2006. Temptor: A lightweight runtime temperature monitoring tool using performance counters. In *Proceedings of the 3rd Workshop on Temperature-Aware Computer Systems*. IEEE, Los Alamitos, CA.

HANSON, H., KECKLER, S., GHIASI, S., RAJAMANI, K., RAWSON, F., AND RUBIO, J. 2007. Thermal response to dvfs: Analysis with an Intel Pentium m. In *Proceedings of the International Symposium on Low-Power Electronics and Design*. ACM, New York, 219–224.

HANUMAIAH, V., VRUDHULA, S., AND CHATHA, K. 2009. Performance optimal speed control of multicore processors under thermal constraints. In *Proceedings of the Design, Automation and Test Conference in Europe*. ACM, New York, 1548–1551.

HE, L., LIAO, W., AND STAN, M. R. 2004. System level leakage reduction considering the interdependence of temperature and leakage. In *Proceedings of the Design Automation Conference*. ACM, New York, 12–17.

HEATH, T., CENTENO, A. P., GEORGE, P., RAMOS, L., JALURIA, Y., AND BIANCHINI, R. 2006. Mercury and freon: Temperature emulation and management for server systems. In *Proceedings of the International Conference on Architectural Support for Programming Language and Operating Systems*. ACM, New York, 106–116.

HEO, S., BARR, K., AND ASANOVIC, K. 2003. Reducing power density through activity migration. In *Proceedings of the International Symposium on Low-Power Electronics and Design*. ACM, New York, 217–222.

HUANG, W., HUMENAY, E., SKADRON, K., AND STAN, M. R. 2005. The need for a full-chip and package thermal model for thermally optimized ic designs. In *Proceedings of the International Symposium on Low-Power Electronics and Design*. ACM, New York, 245–250.

HUANG, W., STAN, M. R., SKADRON, K., SANKARANARAYANAN, K., GHOSH, S., AND VELUSAMY, S. 2004. Compact thermal modeling for temperature-aware design. In *Proceedings of the 41st Annual Conference on Design Automation*. ACM, New York, 878–883.

INTEL. 2002. Intel Pentium 4 processor in the 478-pin package thermal design guidelines. http://www.intel.com/design/Pentium4/guides/249889.htm.

ISCI, C. AND MARTONOSI, M. 2003. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 93–104.

JAYASEELAN, R. AND MITRA, T. 2008. Temperature aware task sequencing and voltage scaling. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, New York, 618–623.

JOSEPH, R. AND MARTONOSI, M. 2001. Run-time power estimation in high-performance microprocessors. In *Proceedings of the International Symposium on Low-Power Electronics and Design*. ACM, New York, 135–140.

JUNG, H., RONG, P., AND PEDRAM, M. 2008. Stochastic modeling of a thermally-managed multi-core system. In *Proceedings of the 45th Design Automation Conference*. ACM, New York, 728–733.

KHAN, O. AND KUNDU, S. 2008. A framework for predictive dynamic temperature management of microprocessor systems. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, New York, 258–263.

KRUM, A. 2000. *Thermal Management*. CRC Press, Boca Raton, FL.

KUMAR, A., SHANG, L., PEH, L.-S., AND JHA, N. 2006. Hybdtm: A coordinated hardware-software approach for dynamic thermal management. In *Proceedings of the 43rd Design Automation Conference*. ACM, New York, 548–553.

KURSUN, E. AND CHER, C. Y. 2008. Variation-aware thermal characterization and management of multi-core architectures. In *Proceedings of the 26th International Conference on Computer Design*. ACM, New York, 280–285.

KURSUN, E., CHER, C.-Y., BUYUKTOSUNOGLU, A., AND BOSE, P. 2006. Investigating the effects of task scheduling on thermal behavior. In *Proceedings of the 3rd Workshop on Temperature-Aware Computer Systems*. IEEE, Los Alamitos, CA.

LI, Y., BROOKS, D., HU, Z., AND SKADRON, K. 2005. Performance, energy, and thermal considerations for smt and cmp architectures. *In Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 71–82.

LI, Y., LEE, B., BROOKS, D., HU, Z., AND SKADRON, K. 2006. Cmp design space exploration subject to physical constraints. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA.

LU, Z., LACH, J., STAN, M. R., AND SKADRON, K. 2005. Improved thermal management with reliability banking. *IEEE Micro. 25*, 6, 40–49.

MCGOWEN, R. 2005. Adaptive designs for power and thermal optimization. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, New York, 118–121.

MONFERRER, P. C., MAGKLIS, G., GONZÁLEZ, J., AND GONZÁLEZ, A. 2005. Distributing the frontend for temperature reduction. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 61–70.

MOORE, J., CHASE, J., RANGANATHAN, P., AND SHARMA, R. 2005. Making scheduling "cool": Temperature-aware workload placement in data centers. In *Proceedings of the Annual Technical Conference*. USENIX, Berkeley, CA, 61–75.

NIMO. 2007. Predictive technology model. http://ptm.asu.edu/.

PILLAI, P. AND SHIN, K. G. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th Symposium on Operating Systems Principles*. ACM, New York, 89–102.

POWELL, M. D., GOMAA, M., AND VIJAYKUMAR, T. N. 2004. Heat-and-run: Leveraging smt and cmp to manage power density through the operating system. In *Proceedings of the11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 260–270.

ROHOU, E. AND SMITH, M. D. 1999. Dynamically managing processor temperature and power. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimization*.

SAMSON, E., MACHIROUTU, S., CHANG, J.-Y., SANTOS, I., HERMERDING, J., DANI, A., PRASHER, R., AND SONG, D. W. 2005. Interface material selection and a thermal management technique in second-generation platforms built on Intel Centrino mobile technology. *Intel Tech. J.* http://download.intel.com/technology/itj/2005/volume09issue01/art06_interface_materials/vol09_art06.pdf.

SKADRON, K., ABDELZAHER, T., AND STAN, M. R. 2002. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 17–28.

SKADRON, K., STAN, M. R., HUANG, W., VELUSAMY, S., SANKARANARAYANAN, K., AND TARJAN, D. 2003. Temperature-aware microarchitecture. In *Proceedings of the 30th International Symposium on Computer Architecture*. ACM, New York, 2–13.

SPRUNT, B. 2002. Brink and abyss Pentium 4 performance counter tools for Linux. Tech. rep., Bucknell University, Lewisburg, PA.

SRINIVASAN, J. AND ADVE, S. V. 2003. Predictive dynamic thermal management for multimedia applications. In *Proceedings of the 17th Annual International Conference on Supercomputing*. ACM, New York, 109–120.

STOER, J. AND BULIRSCH, R. 1991. *Introduction to Numerical Analysis* 2nd Ed. Springer, Berlin.

WANG, S. AND BETTATI, R. 2006a. Delay analysis in temperature-constrained hard real-time systems with general task arrivals. In *Proceedings of the Real-Time Systems Symposium*. IEEE, Los Alamitos, CA, 323–334.

WANG, S. AND BETTATI, R. 2006b. Reactive speed control in temperature-constrained real-time systems. In *Proceedings of the 18th Conference on Real-Time Systems*. IEEE, Los Alamitos, CA.

YANG, J., ZHOU, X., CHROBAK, M., ZHANG, Y., AND JIN, L. 2008. Dynamic thermal management through task scheduling. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE, Los Alamitos, CA, 191–201.

YEO, I., LIU, C. C., AND KIM, E. J. 2008. Predictive dynamic thermal management for multicore systems. In *Proceedings of the 45th Design Automation Conference*. ACM, New York, 734–739.

YUAN, W. AND NAHRSTEDT, K. 2003. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proceedings of the 19th Symposium on Operating Systems Principles*. ACM, New York, 149–163.

ZELDOVICH, N. AND CHANDRA, R. 2005. Interactive performance measurement with vncplay. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*. USENIX, Berkeley, CA.

Received May 2009; revised December 2009; accepted February 2010