DYNAMIC THERMAL MANAGEMENT FOR MICROPROCESSORS THROUGH TASK SCHEDULING

by

Xiuyi Zhou

B.S. in Computer Science, Nanjing University, China, 2002

M.S. in Computer Science, Nanjing University, China, 2005

Submitted to the Graduate Faculty of

Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

PhD in Electrical Engineering

University of Pittsburgh

2011

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Xiuyi Zhou

It was presented on

September 28, 2010

and approved by

Jun Yang, Ph.D., Associate Professor, Electrical and Computer Engineering Department

Alex Jones, Ph.D., Associate Professor, Electrical and Computer Engineering Department

Steven Levitan, Ph.D., Professor, Electrical and Computer Engineering Department

Guangyong Li, Ph.D., Assistant Professor, Electrical and Computer Engineering Department

Rami Melhem, Ph.D., Professor, Computer Science Department

Dissertation Director: Jun Yang, Ph.D., Associate Professor, Electrical and Computer Engineering Department

DYNAMIC THERMAL MANAGEMENT FOR MICROPROCESSORS THROUGH TASK SCHEDULING

Xiuyi Zhou, Ph.D.

University of Pittsburgh, 2011

With continuous IC(Integrated Circuit) technology size scaling, more and more transistors are integrated in a tiny area of the processor. Microprocessors experience unprecedented high power and high temperatures on chip, which can easily violate the thermal constraint. High temperature on the chip, if not controlled, can damage or even burn the chip. There are also emerging technologies which can exacerbate the thermal condition on modern processors. For example, 3D stacking is an IC technology that stacks several die layers together, in order to shorten the communication path between the dies to improve the chip performance. This technology unfortunately increases the power density per unit volumn, and the heat from each layer needs to dissipate vertically through the same heat sink. Another example is chip multi-processor. A chip multi-processor(CMP) integrates two or more independent actual processors (called "cores"), onto a single integrated circuit die. As IC technology nodes continually scale down to 45nm and below, there is significant within-die process variation(PV) in the current and near-future CMPs. Process variation makes the cores in the chip differ in their maximum operable frequency, and the amount of leakage power they consume. This can result in the immense spatial variation of the temperatures of the cores on the same chip, which means the temperatures of some cores can be much higher than other cores.

One of the most commonly used methods to constrain a CPU from overheating is hardware dynamic thermal management(HW DTM), due to the high cost and inefficiency of current mechanical cooling techniques. Dynamic voltage/frequency scaling(DVFS) is such a broad-spectrum dynamic thermal management technique that can be applied to all types of processors, so we adopt DVFS as the HW DTM method in this thesis to simplify problem discussion. DVFS lowers the CPU power consumption by reducing CPU frequency or voltage when temperature overshoots, which constrains the temperature at the price of performance loss, in terms of reduced CPU throughput, or longer execution time of the programs. This thesis mainly addresses this problem, with the goal of eliminating unnecessary hardware-level DVFS and improving chip performance.

The methodology of the experiments in this thesis are based on the accurate estimation of power and temperature on the processor. The CPU power usage of different benchmarks are estimated by reading the performance counters on a real P4 chip, and measuring the activities of different CPU functional units. The jobs are then categorized into power-intensive(hot) ones and power non-intensive(cool) ones. Many combinations of the jobs with mixed power(thermal) characteristics are used to evaluate the effectiveness of the algorithms we propose. When the experiments are conducted on a single-core processor, a compact dynamic thermal model embedded in Linux kernel is used to calculate the CPU temperature. When the experiments are conducted on the CMP with 3D stacked dies, or the CMP affected by significant process variation, a thermal simulation tool well recognized in academia is used.

The contribution of the thesis is that it proposes new software-level task scheduling algorithms to avoid unnecessary hardware-level DVFS. New task scheduling algorithms are proposed not only for the single-core processor, but aslo for the CMP with 3D stacked dies, and the CMP under process variation. Compared with the state-of-the-art algorithms proposed by other researchers, the new algorithms we propose all show significant performance improvement.

To improve the performance of the single-core processors, which is harmed by the thermal overshoots and the HW DTMs, we propose a heuristic algorithm named ThreshHot,

which judiciously schedules hot jobs before cool jobs, to make the future temperature lower. Furthermore, it always makes the temperature stay as close to the threshold as possible while not overshooting.

In the CMPs with 3D stacked dies, three heuristics are proposed and combined as one algorithm. First, the vertically stacked cores are treated as a core stack. The power of jobs is balanced among the core stacks instead of the individual cores. Second, the hot jobs are moved close to the heat sink to expedite heat dissipation. Third, when the thermal emergencies happen, the most power-intensive job in a core stack is penalized in order to lower the temperature quickly.

When CMPs are under significant process variation, each core on the CMP has distinct maximum frequency and leakage power. Maximizing the overall CPU throughput on all the cores is in conflict with satisfying on-chip thermal constraints imposed on each core. A maximum bipartite matching algorithm is used to solve this dilemma, to exploit the maximum performance of the chip.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to show my gratitude to my academic advisor Jun Yang for her continuing support and professional guidance in so many years, and for giving me the opportunity to learn how to do research. I also thank Professor Youtao Zhang in the Department of Computer Science, for the engaging brainstorming sessions that he and Dr. Yang presided. I am professionally indebted to both of them.

I thank Lin Li for being a reliable and skillful colleague to work with. I also feel lucky and happy to work with Yi Xu, Bo Zhao, Ping Zhou, Lei Jiang, Yu Du, Weijia Li, and many others. You have made for a wonderful company throughout my years in Pittsburgh.

Finally, I am heartily thankful to all the professors whose classes I have attended and enjoyed(Steven Levitan, J. T. Cain, Rami Melhem), the dissertation committee members for having contributed to the validation of my work, Alex Jones, Guangyong Li, and the staff in the department and at the Graduate School for making it all possible.

1.0 INTRODUCTION

1.1 THERMAL ISSUES IN CURRENT AND FUTURE PROCESSORS

As technology for microprocessors enters the nanometer era, power density has become one of the major constraints to achievable 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 technology which, for the sake of cost-effectiveness [87], are designed for typical, not worst-case, thermal condition. This entails dynamic thermal managements (DTM) to regulate chip temperature at runtime.

1.1.1 Thermal problem in a single-core processor

With high computational power and high power density, some modern single-core processors, such as Intel Pentium 4 and AMD K6, require heat spreader, heat sink and even cooling fans for faster heat dissipation to the ambient air. Moreover, Pentium 4 processor has two on-chip thermal sensors to monitor the temperatures directly. If the temperature is over some predefined threshold, internal hardware mechanisms are triggered to slow the CPU or even completely power off the CPU [87]. These indicate that the thermal problem has existed for a long time in single-core processors.

1.1.2 Thermal issues in 3D stacked CMP

For the promising 3D integration technology, the situation of thermal issues is even more serious. 3D integration technology is a technology that reduces wiring both within and across disparate dies, as wiring has become a major latency, area and power overhead in modern microprocessors. Studies have shown that wires can consume more than 30% of the power within a 2D CMP(chip multiprocessor) [7]. 3D technology provides vertical stacking of two or more dies with a dense, high-speed interface, reducing the wire length by a factor of the square root of the number of layers used [39]. This significant reduction leads to improved performance and lower power dissipation for the interconnection network. 3D integration technology becomes a promising candidate in constructing future CMP.

One key challenge in 3D die stacking is the heat generation from the internal active layers, because the power density per unit volume increases drastically in 3D. This exacerbates existing hotspots and can create new hotspots within the chip, especially when active logic circuits are vertically aligned. For example, the peak temperature can increase by 17∼20°C in a two-layer 3D implementation for an Alpha-like processor, compared to a 2D design [36, 57]. Other studies on logic-logic stacking 3D floorplans [1, 7, 58] also show similar thermal constraint.

1.1.3 Thermal issues in the CMP impacted by process variation

There is a long-existing problem of process variation(PV) in integrated circuit production. In definition, process variation is the divergence of certain transistor parameters from their nominal values. With the technology size scaling down to 45 nm and below, process variation poses greater challenges for design of future high-performance micro-processors [8], including CMP. Specifically, it makes the cores in a CMP differ significantly in two key parameters: the leakage power each core consumes and the maximum frequency each core can support. These two parameters directly lead to uneven power and thermal distribution across the whole CMP. Without careful planning, an excessive amount of heat can be generated in one

specific area of the CMP, which typically is related to the cores with the highest frequency and leakage, while some other area related to the cores with lower frequency and leakage may remain relatively cool.

1.1.4 Performance losses caused by HW DTM

One of the most commonly used methods to constrain a CPU from overheating is hardware dynamic thermal management(HW DTM), due to the high cost and inefficiency of current mechanical cooling techniques. One example of HW DTM techniques is clock gating. While the CPU is overheated, hardware actions such as clock gating are triggered. Portions of the circuit are disabled so that their flip-flops do not change states. There is no dynamic energy consumption and only leakage current exists. Though global clock gating is a well known power saving technique, it is also used as an effective dynamic thermal management technique. By using this, the temperature of the CPU can be lowered. Other useful HW DTM techniques include but are not limited to dynamic frequency scaling(DFS), dynamic voltage scaling(DVS), issue queue toggling and dynamic voltage/frequency scaling(DVFS). This dissertation mainly alleviates the impact of DVFS on a processor, because DVFS is widely used in high-performance processors for energy saving and temperature constraining. Intel's SpeedStep [85] and AMD's PowerNow! [86] are some industrial implementations using these techniques. Note that we use on-demand clock modulation(ODCM) in the experiment of thermal management on single-core processor, due to the limitation of the Pentium 4 processor. On-demand clock modulation is a unique technique in Intel processor series, which is generally global clock gating(AKA stop-clock). The impact of using ODCM is very similar to DFS.

Currently on real machines with real work loads, HW DTM does not happen frequently, because the current on-chip hardware thermal sensors can not respond quickly. For example, the readings of the sensors on Pentium 4 can only change every second, unable to react to the very fast thermal fluctuation of the die temperature happening in milliseconds. However, the software thermal sensor proposed by Wu et al. [74] can respond very quickly. If such

thermal sensors are used in the processors in future, HW DTM can react to CPU temperature changes fast enough to prevent any thermal overruns. Therefore, in future HW DTMs could be very frequent.

The common side effect brought by HW DTM such as DVFS is the performance loss. When a thermal emergency happens, DVFS is triggered to make the CPU execute fewer cycles in one time unit. Thus, CPU runs at lower speed and the job on the CPU requires longer time to finish. The way to avoid such side effect is basically to avoid the triggering of DVFS. One heavily researched direction is to smartly schedule tasks onto the CPU to prevent the thermal emergencies from happening.

1.1.5 DVFS details

Dynamic voltage scaling [84] is a power management technique, where the voltage used in a component is increased or decreased, depending upon circumstances. Dynamic frequency scaling [83](also known as CPU throttling) is a technique where a processor is run at a less-than-maximum frequency in order to conserve power.

The switching power dissipated by a chip using static CMOS gates is CV^2f, where C is the capacitance being switched per clock cycle, V is voltage, and f is the switching frequency, so this part of the power consumption decreases quadratically with voltage and linearly with frequency. There is also a static leakage current, which has become more and more important as feature sizes become smaller (below 90 nanometres) and threshold voltage becomes lower.

Dynamic voltage scaling is generally done in conjunction with dynamic frequency scaling, at least in CPUs. The speed at which a digital circuit can switch states - that is, to go from "low" to "high" (VDD) or vice versa - is proportional to the voltage differential in that circuit. Reducing the voltage means that circuits switch slower, reducing the maximum frequency at which that circuit can run. This, in turn, reduces the rate at which program instructions that can be issued. It may increase run time for program segments that are sufficiently CPU-bound.

1.2 MOTIVATION AND PROBLEM STATEMENT

The motivation of our work starts from the fact that the future temperature of the processor depends on the current temperature of the processor and the power required for the job running on it. Removing a power-intensive job from the hot processor and replacing it with a low-power job can avoid a potential thermal emergency from happening. On the other hand, when the processor is cool, running a power-intensive job will be thermally safe. This indicates that scheduling tasks in a smart way can avoid DVFS penalties and save performance loss brought on by thermal issues. We explore such opportunities in three different scenarios: a single-core processor; CMP using 3D stacking technology(CMP-3D); and CMP with process variation to maximize throughput(CMP-PV). In each scenario, the common questions are: how are the jobs scheduled to avoid unnecessary thermal emergencies? To what extent can DVFS penalties be lowered and performance loss be saved? In the following section, more details are introduced with regard to the different scenarios.

1.3 THESIS OVERVIEW

As introduced above, we explore such opportunities in three different scenarios: a single-core processor; CMP using 3D stacking technology(CMP-3D); and variation-aware CMP. According different thermal conditions in each scenario, we design specific algorithms or heuristics for task-scheduling policies.

1.3.1 ThreshHot - approaching threshold as close as possible

For single-core CPU, the idea of most of the existing work [14,17,56,27,42,44] is to leverage the distinction between hot and cool jobs, and swap them at appropriate times to control the CPU temperature. We have observed that it is not necessarily best to schedule alternately between hot and cool jobs because cool jobs are precious cooling resources and they should

be used judiciously. Moreover, a job treated as cool in the past may not be necessarily regarded as cool when it is swapped in. This is because future temperature depends on the power of the job and the current temperature as well. The scheduler must determine correctly the temperature slopes for each candidate job to make an informed selection.

We develop a heuristic scheduling algorithm to alleviate the thermal pressure of a processor. Our algorithm achieves this by observing that when the temperature is always below the thermal threshold, executing a hot job before a cool job results in a lower final temperature than execution in a reversed order. Our algorithm outperforms other scheduling algorithms such as one that changes the priority ranks of the hot and the cool jobs [42]. 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 [37, 38, 42], compact thermal modeling [61], and a fast temperature solver [26]. We implemented all these for a Pentium 4 processor.

1.3.2 Power balancing tailored to 3D thermal conditions

To alleviate the exacerbated thermal situation in 3D stacked dies, we propose a heuristic OS-level technique that performs thermal-aware task scheduling on a 3D CMP. Unlike previous thermal-aware OS task schedulers for single core or 2D CMP, our scheduler for 3D chips must take into account the thermal conduction in the vertical direction. Early studies have shown that vertically adjacent dies have strong thermal correlations [2, 75]. For example, a core in one layer could become hot because of a high power task running in the same vertical column but at a different layer. Based on these observations, our proposed scheduler always considers the aggregated power of cores that are vertically aligned. Secondly, we observed the core far away from the heat sink is always hotter than the neighboring one closer to the sink, whatever jobs are assigned to these cores. So we suggest more power-intensive jobs are always put closer to the heat sink. Finally, when a core is overheated, we choose to engage DTM on a vertically aligned core that generates the most power. Such an approach can greatly reduce the total power in one column and quickly cool down the overheated core.

1.3.3 MBM - maximum bipartite matching on CMP-PV

As IC technology nodes continually scale down to 45nm and below, there is significant within-die process variation in the current and near-future CMPs. Process variation(PV) is the divergence of fabrication process parameters from their nominal values. It makes the cores in the chip differ in their maximum operable frequency, and the amount of leakage power they consume. The techniques for tolerating CMP-PV do exist. For example in Intel's Montecito [20], each core has its own clock and V_{dd}, and is called as a voltage and frequency island(VFI). Each VFI contains a clock divider to create its own local clock signal from the output of the shared PLL(phase locked loop). As in AMD's quad-core Opteron [67], asynchronous queues provide interfacing between different clock domains, with the buffers between the cores and their routers implemented as dual-clock FIFOs. To take advantage of the frequency variation of the cores caused by process variation in CMPs, R. Teodorescu et al. [68] proposed an algorithm named *VarF&AppIPC* to map higher-IPC(Instructions Per Cycle) cores to faster cores in order to obtain higher overall throughput. The reason behind this approach is that low-IPC applications are often memory-bound and usually benefit less from high-frequency cores than high-IPC applications do.

Our motivation is to demontrate that *VarF&AppIPC* might not be able to achieve as high throughput as it intends to do, considering the contraint that the CMP must run under a certain thermal limit. In fact, mapping the high-IPC job onto a fast core may exacerbate the thermal condition in that particular core. If DVFS is triggered frequently, the local throughput will be hurt.

The major contribution of this part is that we propose here a task migration algorithm that still tries to maximize throughput but takes both thermal and PV issues into account. The algorithm not only considers the frequency and leakage power information on each core, but also considers the power characteristics of running jobs (tasks). With that information, the algorithm predicts the throughput of each core-job binding, and uses the Maximum Bipartite Matching algorithm (abbreviated as MBM in this thesis) to get the optimal mapping.

1.4 CONTRIBUTIONS

In summary, the contributions of this thesis are as follows:

- In a single-core, a heuristic algorithm named ThreshHot judiciously schedules hot jobs before cool jobs, to make the future temperature lower. Furthermore, it always makes the temperature as close to the threshold as possible while not overshooting.
- In 3D stacked processors, three heuristics are proposed. First, vertically stacked cores are treated as a core stack. The power of jobs is balanced among the core stacks instead of individual cores. Second, hot jobs are moved close to the heat sink to expedite heat dissipation. Third, when thermal emergencies happen, the most power-intensive job in a core stack is penalized in order to lower the temperature quickly.
- In the CMPs affected by significant process variation issues, maximizing the throughput is in conflict with on-chip thermal constraints. A maximum bipartite matching algorithm is used to solve this dilemma, and the implementation details are discussed.

All the algorithms proposed try to avoid thermal emergencies and the subsequent hardware DVFS. As a result, the chip performance is maintained.

1.5 ROADMAP

The rest of the thesis is organized as follows. Chapter 2 presents an introduction of power obtaining and temperature prediction. Chapter 3 introduces prior work in three scenarios we investigate. The proposed scheduling policies are explained in Chapter 4. Chapter 5 concludes and describes the future work.

2.0 OBTAINING POWER AND TEMPERATURE

All of our thermal-aware scheduling algorithms in this dissertation need information about the peak temperature of the processor. Also, the power consumed for the executed jobs needs to be obtained. In this chapter, we explain how these values can be obtained or estimated at runtime.

2.1 TEMPERATURE OBTAINING AND COMPUTATION

2.1.1 Thermal sensor readings are insufficient.

It seems that the OS could leverage existing on-chip thermal sensors for temperature readings. Unfortunately, this is insufficient because, in addition to the current temperature, our algorithm also needs to predict 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.

2.1.2 Temperature model.

Consider the simplified thermal model for a processor treated as a single node. The duality between heat transfer and electrical phenomena [41] has provided a convenient basis for modeling the chip temperature using a *dynamic compact thermal model* [61]:

$$\frac{1}{R}T + C\frac{dT}{dt} = P, \tag{2.1}$$

where T is the temperature relative to the ambient air. R and C are effective vertical thermal "resistor and capacitor" for the entire chip. 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. $\frac{dT}{dt} \neq 0$).

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:

$$\mathbf{G}T + \mathbf{C}\frac{dT}{dt} = P\ , \tag{2.2}$$

which is the matrix form of Equation (2.1) with $\mathbf{G}$ being the matrix of the thermal conductances. Both T and P are now vectors. Therefore, to obtain the temperatures in the next time interval for a candidate job interval, the scheduler must solve Equation (2.2) 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 thermal management threshold.

2.1.3 Temperature calculation speedup.

Equation(2.2) may seem like a lot of computation for the scheduler to solve at runtime. Fortunately, previous work has shown that the complexity of Equation (2.2) can be greatly reduced if the time interval Δt is kept constant [26]. This is the case in our scheduler. Here we discuss the concept briefly.

The linear system in equation (2.2) has a complete solution as:

$$T(t) = e^{\mathbf{C}^{-1}\mathbf{G}t}T(0) + \int_0^t e^{\mathbf{C}^{-1}\mathbf{G}(t-\tau)}\mathbf{C}^{-1}P(\tau)d\tau \tag{2.3}$$

For a fixed length scheduling interval Δt, we take the average power during the interval so that $P(t)$ can be factored out. (2.3) is now:

$$T(\Delta t) = \mathbf{A}T(0) + \mathbf{B}P \tag{2.4}$$

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 the linear system (2.4) is time-invariant, it holds for every interval Δt. Therefore:

$$\begin{aligned} &T(n\Delta t) = \mathbf{A}T((n-1)\Delta t) + \mathbf{B}P(n-1), \quad \text{or simply} \\ &T(n) = \mathbf{A}T(n-1) + \mathbf{B}P(n-1) \end{aligned} \tag{2.5}$$

As we can see, once $\mathbf{A}$ and $\mathbf{B}$ are pre-calculated and stored, the temperature at any step n can be found through a 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. For example, a P4 thermal model has 82 nodes in total and computing the 82×1 temperature vector at runtime takes only $\sim$16.45μs. For CMP-3D or CMP-PV, where the core numbers are much larger, the computation overhead of temperatures can be distributed to all the cores, assuming the lateral heat dissipation among cores can be ignored [80]. Next, we will discuss how to obtain the power values $P(n-1)$ online.

2.2 COMPUTING THE POWERS

2.2.1 Power estimation

Recent research has proposed to incorporate on-chip power sensors for power and thermal control [49]. 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 [5, 37, 38]. 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 work 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 [34]. Since we also calculate the temperature online, we consider the leakage as an integral part in our power estimation. In the scenarios of single-core processor and CMP-3D we adopted a model developed in [29, 45] using PTM(Predictive Technology Model) 0.13μ technology parameters [90], matching the processor technology size(Pentium 4 Northwood) in our experiment. We determined the necessary device constants through SPICE simulation. In the scenario of CMP-PV, the core-wise leakage variation is exposed by the effective leakage parameter, p_{eff} [32]. PTM $32nm$ technology node is used in SPICE simulation to find p_{eff} in each individual core. In practice, such core-wise leakage variation parameters can be provided by the chip manufacturers during post-manufacturing.

2.2.2 Power prediction

The last issue we need to resolve is the prediction of power consumption of a job in the *next* scheduling interval, as required by Equation (2.5). 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 1. 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 4.1.4) did not show significant disadvantages in most cases, indicating that (at least in those cases) mispredictions did not lead to much mis-scheduling. For example, if a job will have the i^{th} highest temperature among all candidates, and even if the power is mispredicted, its predicted temperature will still remain in the i^{th} position, then power misprediction does not change the scheduling decision. This is observed clearly in *crc* and *md5*, which tend to alternate between two different power levels. Hence, the last-value predictor always missed the right value, but the error does not lead to big temperature changes, and therefore, did not impact the scheduling decision.

2.3 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

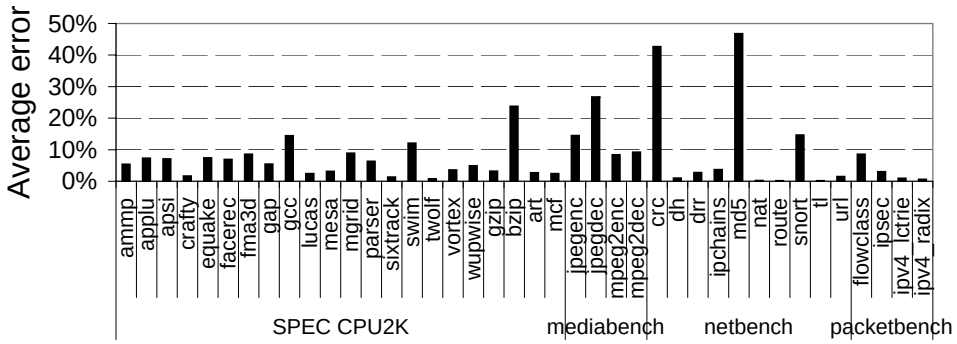


Figure 1: Average error rates for last power value predictor.

interval. They are then converted into the power consumption to 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 for next job selection. The flow is depicted in Figure 2(a).

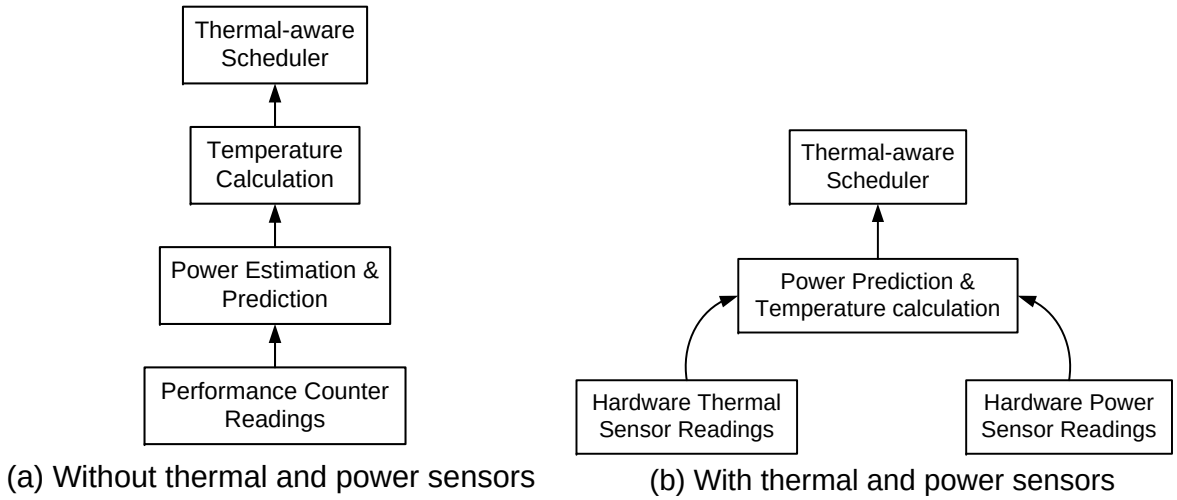


Figure 2: Thermal-aware task scheduling methodologies.

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 2(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 online self-calibration to lessen the error due to thermal and power model abstraction.

2.4 THE ACCURACY OF TEMPERATURE CALCULATION

W. Wu et al. in [74] developed such a software thermal sensor (STS) in a Linux system with a Pentium 4 Northwood core, using the same methodology as shown in Figure 2(a). Their power and thermal models were calibrated and validated against on-chip sensor readings, thermal images of the Northwood heat spreader, and the thermometer measurements on the package. The thermal profiles they obtained through a continuous execution of some SPEC2K benchmarks for hours show that the computed temperatures on the heat spreader using STS are very close to the thermometer readings on the heat spreader. The reason for the discrepancy between the on-die sensor readings and the computed temperatures of 24 function units is that the sampling interval of the on-die sensor on P4 Northwood is fixed at one second, while the STS can compute temperatures every eight milliseconds. Intel admits the sensors are slow. Therefore, the on-die sensor can miss lots of thermal details. However, W. Wu et al. in [74] show two achievements of STS at least. One is that there is a similarity between the thermal changes from on-die sensor readings and the computed ones. The other is that the errors due to modeling and abstraction do not propagate. Finally, to make sure the temperatures of the function units are accurate, they also calibrated them with the results obtained from the micro-architecture level thermal simulation tool - HotSpot developed by Skadron et al. [62].

3.0 RELATED WORK

Other researchers have done some work in the three thermal scenarios we deal with. However, for each case, our work has improved upon their work or has significant differences from theirs. Here we list the prior work and compare those projects with ours.

3.1 PRIOR WORK IN A SINGLE CORE PROCESSOR

First we introduce the related work on single-core processors. Some recent work has developed temperature control techniques for *real-time* workloads [3,4,69,71]. 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 approaches can be used to minimize the energy consumption for real-time systems as well [55, 77]. 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

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

to *balance* the chip temperature at runtime [56]. In another work [17], 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 [14]. All these approaches exploit a simple interleaving between hot and cool jobs when it comes to scheduling. 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 thesis, we demonstrate this philosophy using a scheduling heuristic on a single-core processor, and leave its extensions to CMPs as future work.

In the single-core domain, the "HybDTM" [42] 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 [6] where the energy dissipation rate is evened among hot and cool jobs by 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 [52], a temperature-aware workload placement heuristic was studied for data centers to minimize the cost of cooling. The "Mercury and Freon" [30] framework uses software to estimate temperatures for a server cluster and manages its component temperatures through a thermal-aware load balancer. The "ThermoStat" [13] 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.

3.2 PRIOR WORK IN CMP BUILT WITH STACKED DIES

There have been many works recently investigating the performance potential and the challenges in 3D CMP designs. Mysore et al. [54] proposed to stack on top of a normal processor a profiling die that can identify memory leakage, perform diagnosis etc. to save the area and power on the main die. Black et al. [7] studied the performance advantages and thermal challenges for stacking a large DRAM and SRAM cache on a processor, as well as implementing a processor in two layers. Xie et al. [75] reported that the peak temperature in a 3D chip of 2 layers and one die per layer can be as high as 125°C. More importantly, there is only a difference of a couple of degrees, in the worst case, between the hotspots in the top die and the bottom die. This indicates a strong thermal correlation among adjacent layers in a 3D processor. To ensure better heat dissipation in a 3D chip, Puttaswamy et al. proposed a "Thermal Herding" design [58] at the micro-architecture level which lowers the power of the chip by splitting individual function unit blocks across multiple layers, and places the most frequently switched part, or activity, closest to the heat sink. Alternatively, adding thermal vias can also alleviate the thermal conditions within a 3D chip. Goplen et al. [24] studied that proper placement of thermal vias in 3D IC design can obtain a maximum of 47.1% reduction in temperature. In the multicore domain, Loh et al. [48] introduced different approaches for implementing single-core and multicore 3D processors. Particularly, they pointed out that stacking separate cores (in multicore design) can significantly reuse the existing 2D designs, and the interface between the cores needs no more than a few thousand connections.

Compared to the previous work, this thesis focuses mainly on software approaches to thermal management in 3D CMP. There have been proposals on OS-assisted thermal management for single core chips. The HybDTM [42] technique controls temperature by limiting the execution of a hot job once it enters an alarm zone. This is achieved by lowering the priority of the hot job so that the OS allocates fewer timeslices to it and gives cool jobs relatively more timeslices to execute. An ideal simulation study was performed in [44] to show the

benefits of interleaving hot and cool job executions. However, neither performance study nor task switching overhead was considered. In the 2D multicore domain, the "Heat-and-Run" scheduler [56] assumes that there is always an idle and cool core present in a CMP such that an overheated core can migrate its thread to the cool core. However, in our technique, all cores are assumed to be busy and have temperatures above an idle core's temperature. Choi et al. [14] compared and implemented three different task schedulers, heat-balancing, deferred execution, and threading with cool-loops, to leverage temporal and spatial heat slacks among application threads. The proposed mechanisms are implemented in the PowerPC5. Chong et al. [66] proposed a 3D MPSoC thermal optimization algorithm that conducts task assignment, scheduling, and voltage scaling for a set of real-time workloads. The goal was to slow down the workloads as long as the deadlines are met. This is quite different from our approach which focuses on best performance and low thermal profile.

3.3 PRIOR WORK IN CMP WITH PROCESS VARIATION

Borkar et al. [8] point out that process variation could be one major challenge in near-future chip production and examples of PV modeling can be seen in [10, 59]. In these papers, the PV effects, both random and systematic, within-die and die-to-die variations are modeled. The impact of PV on the performance and leakage power of logic and SRAM structures is evaluated. Our thermal-aware task scheduling algorithm is based on their work and especially relies on the modeling tool provided by Sarangi et al. [59].

Brooks et al. [11] define and investigate the major components of any hardware dynamic thermal management scheme. Skadron et al. [62] provide a compact thermal model and a fast simulation tool for evaluating the impact of dynamic thermal management. For software-level thermal management by task migration, there are exemplar ideas proposed in [22, 14, 44, 42].

More recently, there have been papers that take power/thermal management along with PV into consideration. Kursun et al. [43] use thermal imaging technique to sense the process

variation of the CMP. Their methodology is to treat the CMP with PV as a "black-box", while our work models PV from the ground up. Moreover, they only consider leakage variation and don't model the possible frequency differences of the cores. Wang et al. [70] accommodate process variation information by using a coefficient matrix, and apply optimal control theory to keep the power and temperature within constraints. The difference between their work and ours is that they haven't considered the potential opportunities brought by task migration. Teodorescu et al. [68] and Herbert et al. [32] model the frequency and leakage discrepancies on CMP from ground up. Teodorescu et al. [68] attempt to maximize the throughput by heuristically mapping the CPU-intensive jobs onto the faster cores and mapping memory-intensive jobs onto the slower cores. Such a fixed mapping may generate sub-optimal results when the faster cores trigger DVFS during thermal emergencies. By setting local and total power constraints, they expand their *VarF&AppIPC* algorithm and use linear programming to maximize the performance. Their linear programming algorithm will change back to *VarF&AppIPC* when the CMP is under thermal constraints, because the sum of the temperatures of the cores is not necessarily below a limit. Each core only needs to satisfy its own thermal constraint. Herbert et al. [32] design a DVFS scheme that adapts to each core's own frequency and leakage settings. Their goal is to minimize the energy per instruction and maintain the throughput of the CMP. The difference of the idea in [68, 32] from ours is that they both bind high-IPC jobs onto faster cores to take advantage of the high frequency. We demonstrate this is not an optimal solution with the thermal constraints considered, and propose a new algorithm.

4.0 PROPOSED TASK SCHEDULING SOLUTIONS

Here we discuss about our detailed task scheduling policies on three different thermal scenarios. In each section, we first introduce our idea; then we present specific algorithms or heuristics, which implement the idea. Finally, the results are shown at the end of each section.

4.1 THRESHHOT

When a modern single-core processor heats up due to high power, a dynamic thermal management technique curbs the overshooting of die temperature. However, the CPU suffers from performance loss. In the following, we design a heuristic task scheduling method to prevent the temperature from trespassing into this thermal threshold. In this way, the triggering of DTM can be reduced and performance loss can be avoided.

4.1.1 Thermal scheduling algorithms

When the processor underlying the OS 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, etc. Thus, independent 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.

4.1.1.1 The principle To keep the temperature below the threshold, a naïve, greedy algorithm tries to control the temperature by keeping the *current* chip temperature as low as possible, 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, that is the greedy schedule) then the temperature at the end of $t+1$ is higher than for the order yx (y before x).

Consider the simplified thermal model for a processor treated as a single node. The duality between heat transfer and electrical phenomena [41] has provided a convenient basis for modeling the chip temperature using a *dynamic compact thermal model* [61]:

$$\frac{1}{R}T + C\frac{dT}{dt} = P, \tag{4.1}$$

where T is the temperature relative to the ambient air. R and C are the effective vertical thermal resistor and capacitor of the entire chip. 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 (4.1) can be approximated by

$$\frac{1}{R}T_i + C\frac{T_i - T_{i-1}}{\Delta t} = P \tag{4.2}$$

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:

$$T_n = \alpha^n T_0 + (\alpha^{n-1} + \alpha^{n-2} + \cdots + 1)\beta P \tag{4.3}$$

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

$$T_{2n}^{xy} = \alpha^{2n} T_0 + (\alpha^{n-1} + \alpha^{n-2} + \cdots + 1)\beta(\alpha^n P_x + P_y) \tag{4.4}$$

For schedule yx, this final temperature will be

$$T_{2n}^{yx} = \alpha^{2n} T_0 + (\alpha^{n-1} + \alpha^{n-2} + \cdots + 1)\beta(\alpha^n P_y + P_x) \tag{4.5}$$

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.* Figure 3 gives an intuitive illustration of the impact on temperature with different scheduling order. 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.

Suppose now you are given (offline) a collection X of job intervals, each with known power consumption, and that they can be executed in some order without exceeding the threshold.

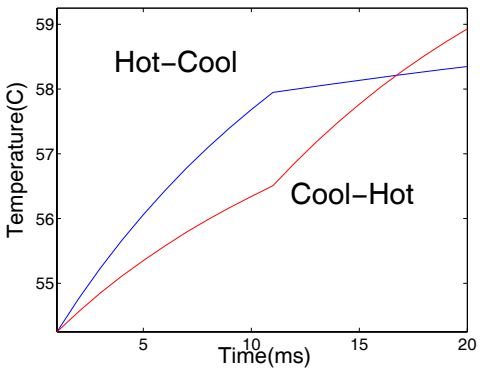


Figure 3: The impact of scheduling a hot and cool program in different orders.

Suppose further that in this order there are two consecutive job intervals x, y with x before y, such that $P_x < P_y$ and that executing x first will not exceed the threshold. Then, by the argument above, we can exchange x, y and the temperature in the new schedule will still stay below the threshold. 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 later in the schedule. By repeating this argument, we obtain a schedule of X that follows the consequent policy $\mathcal{P}$: *at each step choose the hottest job that will not increase the temperature above the threshold.*

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 temperatures. 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 4.1.2). Thus, in addition to the rules described above, the choice of the next job to run must be consistent with these fairness restrictions.

4.1.1.2 In practice Early discussion has assumed a simple case where the CPU is considered as a single node and the heat is only dissipated through a 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 (4.1) will be expanded into a matrix form in which every node is described as:

$$\frac{T-T_1}{R_{L1}}+\frac{T-T_2}{R_{L2}}+\frac{T-T_3}{R_{L3}}+\frac{T-T_4}{R_{L4}}+\frac{T}{R}+C\frac{dT}{dt}=P \tag{4.6}$$

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 i^{th} 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 (4.6), is higher than the T_i's. Also, heat is removed mostly from the vertical path and less from the surface [17, 56, 61]. 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 (4.6) is dominated by the last two terms, we can still treat a hotspot as a single node as before.

4.1.2 Linux kernel implementation

To evaluate our thermal-aware scheduling policy, we implemented all the modules in Figure 2(a) into a Linux kernel version 2.4.18 with O(1) scheduler patch. The major challenge was how 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 [9] and then describe our modification.

4.1.2.1 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 a 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 finish 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.

4.1.2.2 Our modification The execution of a time quantum is periodically interrupted by the kernel's interrupt handler, typically once every 1-10ms. 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*.

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 work assumed 10ms as the RC constant of the hottest unit on a CMP processor [17, 56]. 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 requirements(e.g., the counter rotation effect [64]), 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.

In the original scheduler, jobs can occupy the processor for its entire time quantum. For batch jobs, the default time quantum is $100ms$ [9]. 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 $\sim 35.35\mu s$ 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 $8ms$. This reduces total context switches and still results in an overall performance gain.

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.

Recall that we intend to apply our thermal-aware policy only to the batch jobs; but we still need to consider the possible impact on real-time and interactive jobs. Batch and interactive jobs are given a different range of priorities(100-140) than the real-time jobs(0-99). The candidate jobs that are eligible for thermal scheduling fall within the batch job's priority range. This ensures that our scheduler does not touch any real-time jobs and they are scheduled in the same way as before. Although interactive jobs share the same priority range with batch jobs, Linux implements the TASK_INTERACTIVE macro based on the past behavior of the job to decide whether a job should be considered as interactive or batch. Our scheduler implementation can use the macro to bypass those jobs considered as "interactive" by the operating system. We experimented with a GUI application `VNCplay`

developed by Zeldovich *et al.* [78]. This program records user's interactive activities such that it can be replayed multiple times. We recorded some user activities by playing a TIC-TAC-TOE game and editing a file using vim editor. Figure 4 plots the cumulative distribution function of the interactive latencies for our recorded user events. *Baseline* means the Linux baseline scheduler. *ThreshHot* is our thermal-aware scheduler. And we further tested *ThreshHot* with and without some batch jobs running in the background. Figure 4 shows for any percentage of user events shown in y axis, the response always happpens in seconds. For example, 70% of the user events get response in 2 seconds, in all three scenarios. This proves our thermal-aware scheduler does not show any noticeable impact on the interactive jobs.

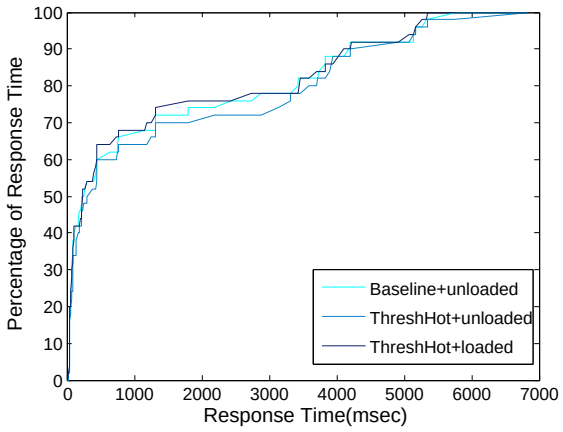


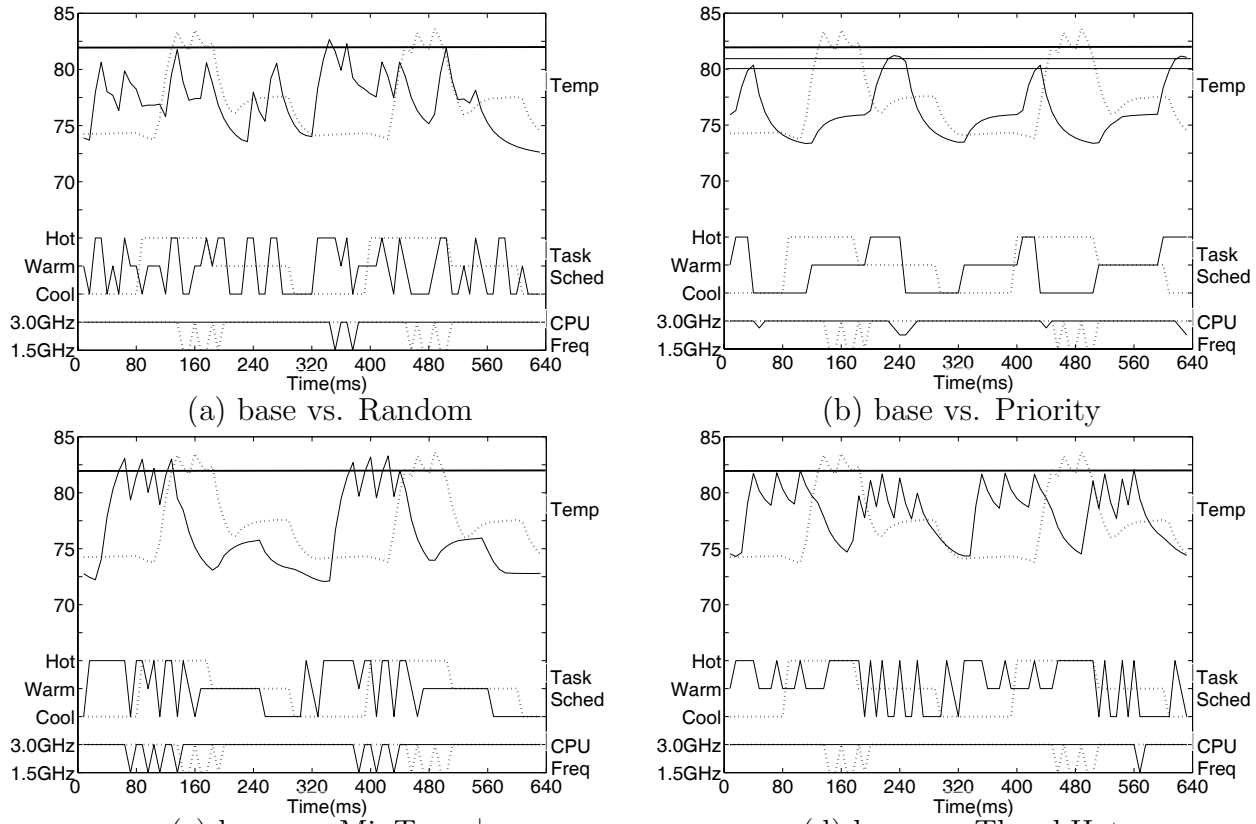
Figure 4: Variation in latencies for VNCplay in our thermal-aware scheduler.

4.1.3 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, and tested the scheduling algorithms below 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 [42]. A job is considered "hot" if its overall temperature in an epoch exceed a pre-defined soft threshold which is lower than, but close to, the hardware threshold. The priority is adjusted proportional 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 [44]. 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 6 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 5 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



(a) base vs. Random

(b) base vs. Priority

(c) base vs. MinTemp$^+$

(d) base vs. ThreshHot

Figure 5: 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.

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 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.

4.1.3.1 Random scheduler As we can see from Figure 5(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 scaling events 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.

4.1.3.2 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 per epoch compared to cool jobs, the cool jobs make more progress and may eventually finish earlier than the hot jobs. As we can see from Figure 5(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.

4.1.3.3 Mintemp$^+$ scheduler This scheduler tends to oscillate between the hottest and the coolest job, as shown in Figure 5(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 4.1.4 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.

4.1.3.4 Threshot 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 5(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.

4.1.4 Experimental evaluation

Unlike in some previous work, 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 [33,34,61] 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).

4.1.4.1 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 6 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.

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 6, 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

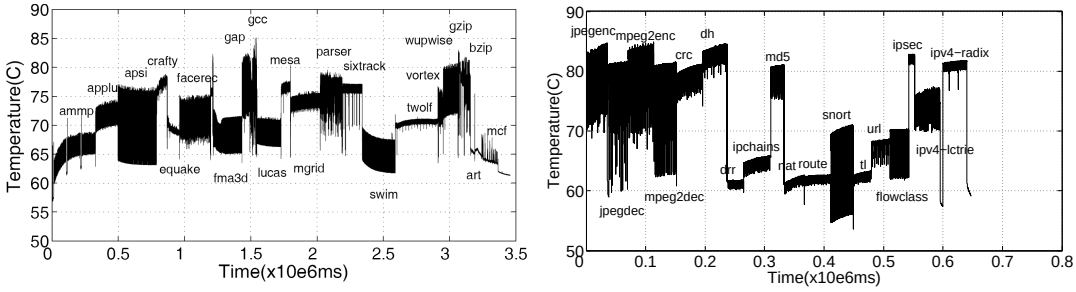


Figure 6: Thermal profiles of the IntReg for all 22 SPEC2K (left) and media, net, and packetbench (right).

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.

Table 1: 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

4.1.4.2 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.

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

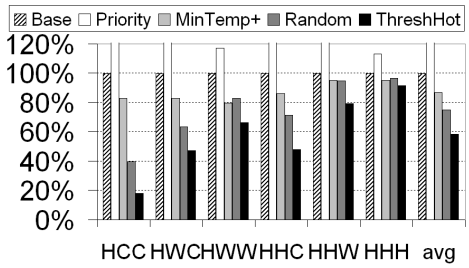
	SPEC2K	**media, packet and netbench**
HHH	gzip sixtrack vortex	jpegdec ipv4_lctrie md5
HHW	mgrid gzip bzip	jpegenc jpegdec flowclass
HHC	gzip bzip art	mpeg2enc mpeg2dec tl
HWW	gap apsi twolf	ipv4_lctrie url ipchains
HWC	gcc apsi art	ipv4_radix ipchains nat
HCC	mesa ammp mcf	dh drr route

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.

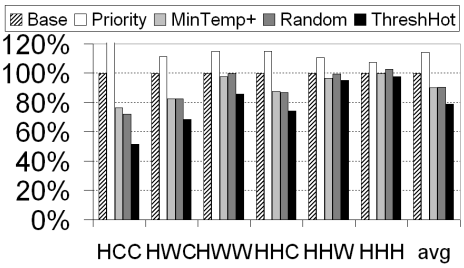
To test the sensitivity of different schedulers to different environmental conditions, we varied the frequency scaling threshold from 75°C to 73°C and 71°C (from Figure 6, most programs' steady temperatures range between ∼60°C and ∼80°C). With a steady test-environment temperature (26°C in our case), 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 [87] 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.)

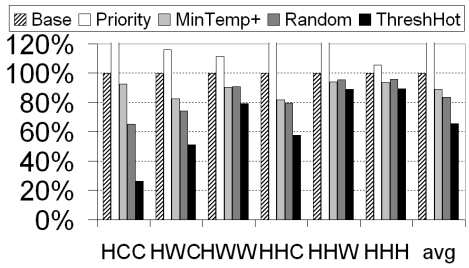
(a) Mild thermal environment



(b) Harsh thermal environment



(c) Medium thermal environment



(d) Mix of mediabench, packetbench, and netbench in medium environment

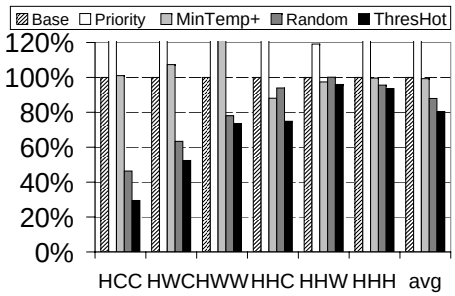
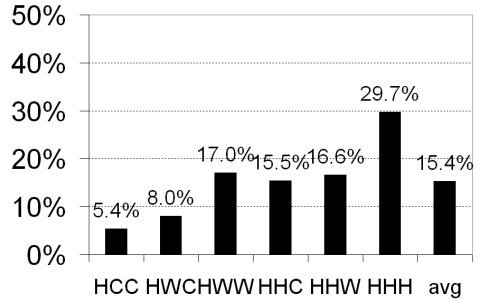


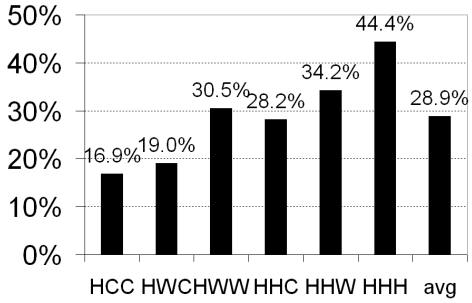
Figure 7: Number of thermal emergency triggers, normalized to the baseline scheduler (Linux default).

4.1.4.3 DTM reductions Figure 7 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.

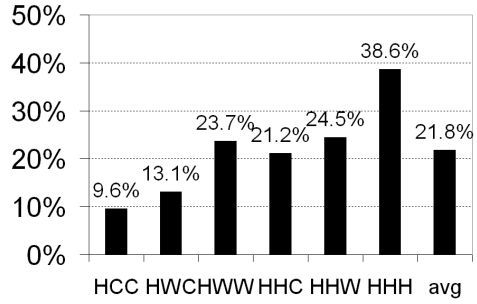
(a) Mild thermal environment



(b) Harsh thermal environment



(c) Medium thermal environment



(d) Mix of mediabench, packetbench, and netbench in medium environment

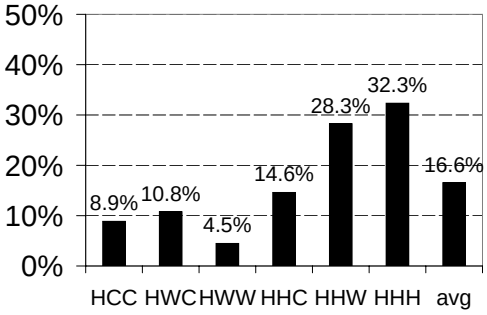
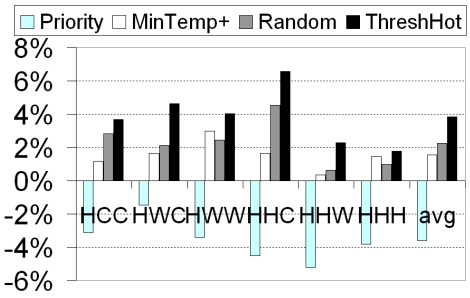
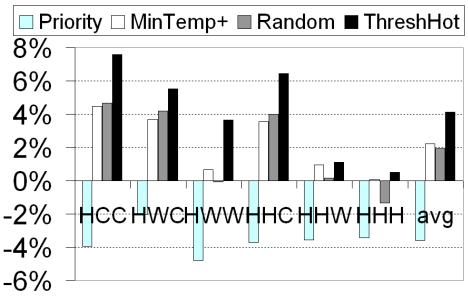


Figure 8: Percentage of execution time under DTM in the baseline scheduler.

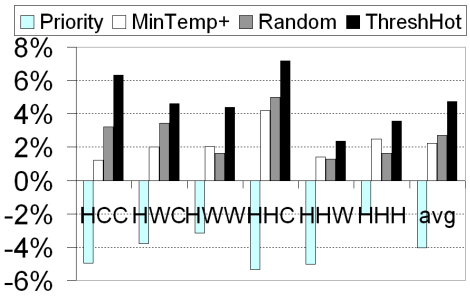
(a) Mild thermal environment



(b) Harsh thermal environment



(c) Medium thermal environment



(d) Mix of mediabench, packetbench, and netbench in medium environment

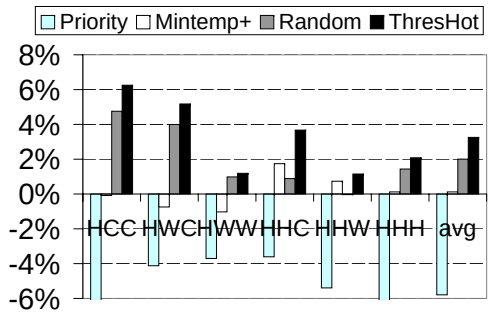


Figure 9: The percentage of the execution time reduction from the baseline.

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 7(c)), the MinTempPlus scheduler reduced DTMs in the baseline schedule by 7.5%, the Random scheduler reduced 34.7%, while the ThreshHot scheduler reduced big as much as 73.6%.

The Random scheduler performs slightly better than the MinTempPlus 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 7(b) and (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 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 cannot be seen from Figure 7(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.

4.1.4.4 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 8 plots the percentages of execution time attributed to DTMs. Figure 9 shows the overall performance improvements. The three subgraphs represent different thermal environments, similar to Figure 7. As expected, the ThreshHot scheduler consistently and significantly outperforms other schedulers. The Priority scheduler brings negative impact on performance unless there is constant supply of cool jobs, which was assumed in the original work [42]. 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 Figures 7 and 8 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 8(a)), the ThreshHot scheduler can easily remove 81.9% of them (Figure 7(a)). When the DTMs occur 44.4% of time in 'HHW' (Figure 8(c)), the ThreshHot scheduler can only remove 2.5% of them (Figure 7(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 9 shows the overall performance improvement, reflecting both the reduction of DTMs from Figure 7 and the original number of DTMs produced by the base scheduler, as seen in Figure 8. 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 the 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%, and 3.25% for mild, medium, harsh thermal environment, and non-SPEC programs respectively.

We also observed that the MinTempPlus 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 MinTempPlus, 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.

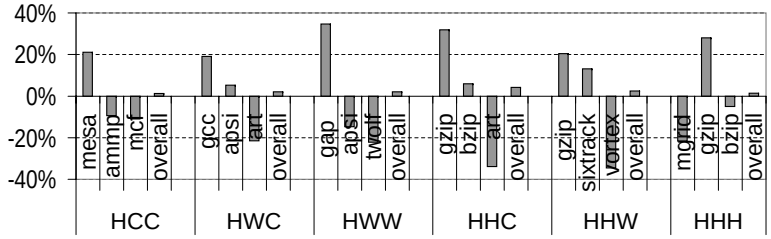


Figure 10: Drastic performance changes to individual jobs by MinTempPlus scheduler (mild thermal environment).

One important downside of the MinTempPlus 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 10, 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.

4.1.4.5 Overhead The overhead of our ThreshHot scheduler (and MinTempPlus and Priority) mainly comes from the temperature calculation in the scheduler and the 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 module working in the kernel for a sufficiently long time. This overhead includes probing the hardware performance counters, calculating power and calculating the temperatures using the method described in Section 2.1. As we also mentioned in Section 4.1.2.2, 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. Figure 11 shows the details of the overhead by running SPEC2K benchmark workloads in the medium themal environemnt. The overhead is normalized to the total execution time of Baseline. On the average, MinTempPlus and ThreshHot incur the overhead of 1.18% and 1.22% respectively, in which 0.77% and 0.70% are from temperature computation. 0.62% out of 0.66% of the overhead incurred by Priority is also from temperature computation. Although Random incurs 0.5% task switch overhead on the average, it does not have the overhead of temperature computation.

4.1.4.6 Impact of varied intervals on ThreshHot Although our 8ms scheduling interval is very close to the minimum interval recommended by Linux, other operating systems may have a different requirement for the interval length. Our ThreshHot scheduler works well when the scheduling interval is 8ms. When the interval increases, some hot jobs can easily raise the temperature above the threshold. The warm jobs start to have similar ther-

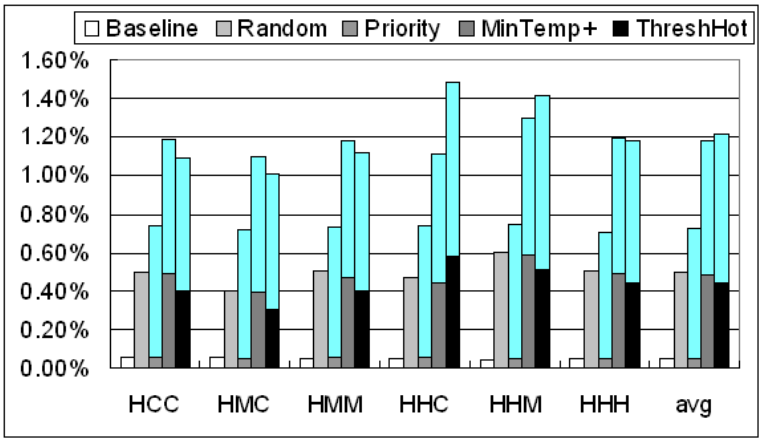


Figure 11: Details of the time overhead(represented in percentage in y-axis) incurred by the temperature computation and task switching(upper area marks the overhead of temperature prediction), normalized to the execution time in Baseline in the medium thermal environment.

mal behavior as the hot jobs do in $8ms$, because the warm jobs can make the temperature much closer to the threshold in an interval longer than $8ms$. So the cool jobs are now used to cool the temperatures raised by the warm jobs, leaving hot jobs unattended. We did experiments by using the intervals of $16ms$ and $32ms$. As Figure 12 shows, The benefits obtained by ThreshHot decrease significantly when the interval becomes longer. On the average, ThreshHot achieves 3.21% performance improvement over Baseline when the interval is $16ms$, and achieves only 2.06% improvement when the interval is $32ms$.

4.1.4.7 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 earlier in Section 2.2.2, 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 13 has shown the percentages of errors in predicted power values using the last

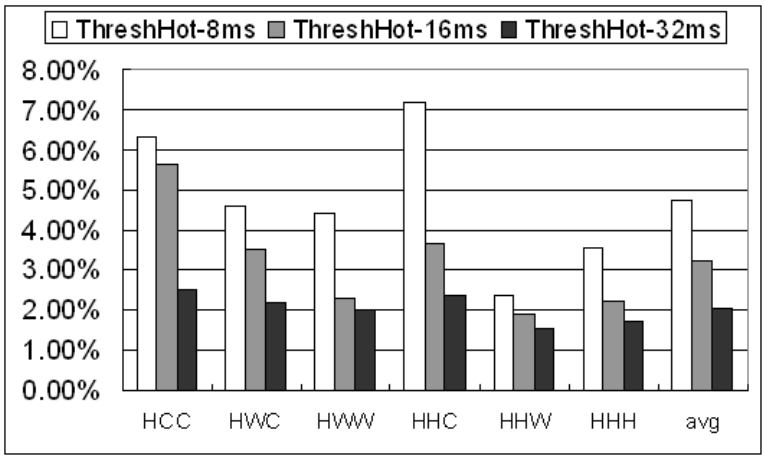


Figure 12: The relative performance improvement by ThreshHot over Baseline, under different scheduling intervals.

value prediction. In this section, we quantify the impact of such errors on performance improvement, to justify the usage of the last value power predictor in ThreshHot scheduler.

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 scheduling with the last value power predictor and another time with the oracle predictor. In our scheduler, the power predictor works with the scheduler in the following way. 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, the temperatures are searched from high to low to 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 select 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.

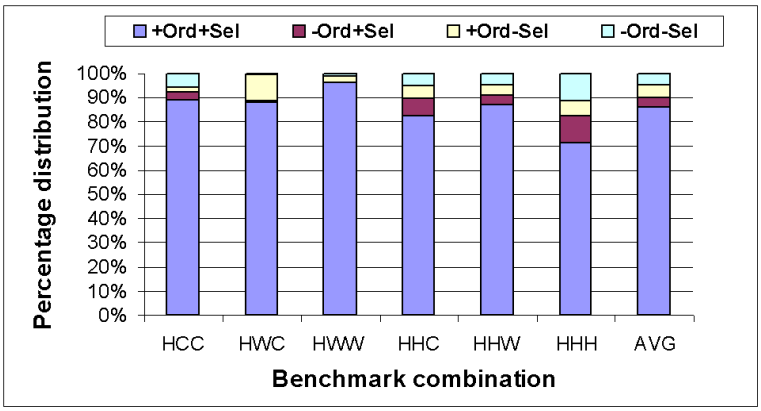


Figure 13: The distribution of last value prediction results.

Figure 13 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 14 further shows the performance speedups for the last value predictor and the oracle power predictor. As we can see, on average, the last value predictor achieves only 0.6% less speedup 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.

4.1.4.8 Scalability One of the concerns of the scheduling overhead is whether the algorithm can be scaled up to support more jobs. With a large number of jobs, more time

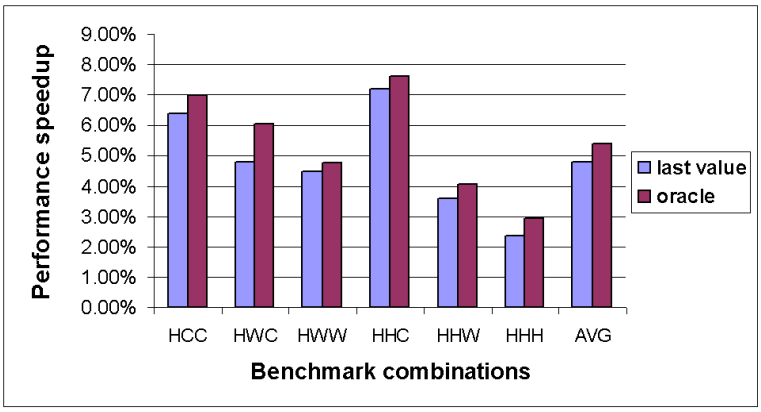


Figure 14: The offline performance comparison of last_value power predictor and oracle power predictor

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 time complexity of the sorting is $O(N \log N)$. However, this takes a short time when N is small. 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 15.

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 4 jobs and a minimum of 0.25% for 7 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 have explained above, 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 1 and 4 jobs. The actual time depends on specific temperature values of different jobs. That is, more 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 4.1.4.2 were for scheduling 3 jobs. As we can see here that the scheduling overhead for 3 jobs is around the average. Therefore, we conclude that our proposed ThreshHot is a scalable solution. For future CMPs, the number of jobs will increase proportionally with the number of cores. Suppose there are 64 cores and 300+ jobs, each core will then be assigned around 5 jobs in its local job queue. The question here is how to assign jobs to each core in order to make cores balanced in their thermal behaviors. One possible solution is to sort the jobs according to their power history and then try to make sure each core has a balanced number of hot and cool jobs. This sorting and the following

job migration could be done every several seconds, to keep the overhead as small as possible. This could be left as our future work. However after the job assignment is done, each core will still only incur 1% overhead compared with Random because the local job queue has only 5 jobs. If using our ThreshHot algorithm each core can gain 5% average performance improvement, it's still worthwhile to employ our scheduling mechanism.

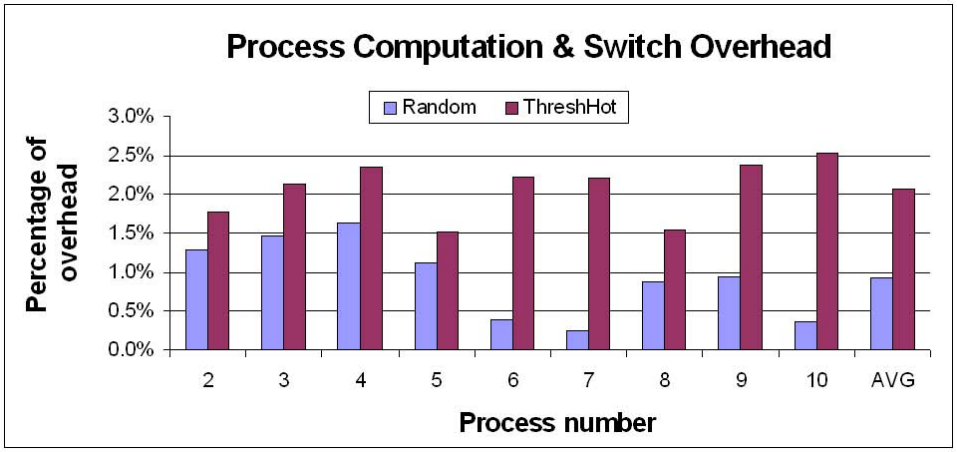


Figure 15: The overhead from context switch and temperature computation(x-axis shows the number of processes)

4.2 BALANCING BY STACK IN 3D CMP

CMP with dies stacked is a promising technology to reduce wiring overhead in the layout. However, the stacking of logic layers can generate more heat and the heat may exceed the chip cooling capacity. In the following, we first explore the thermal characteristics of 3D CMP. The thermal characteristics then motivate us to add three heuristics to the task scheduling policy.

4.2.1 Motivation and rationale

In this section, we analyze the thermal characteristics of 3D CMP as achitectures, by looking at a sample of proposed floorplans. Then, we focus on exploring the thermal characteristics of that particular floorplan.

4.2.1.1 A representative floorplan There have been a number of 3D CMP floorplans, as shown in Figure 16 (a)-(c), proposed in literature [1, 7, 48]. In these figures, cores are stacked on each other, with extended cache or memory in between. We observed that for a 3D stacked chip to be scalable in layer count, it is inevitable to encounter more than one active cores in one vertical core column, no matter how the active cores and cache banks are placed in the floorplan. Further, if we look at the distance of each core stack to the heatsink (on either the top or bottom of the chip), we can classify these floorplans into two categories.

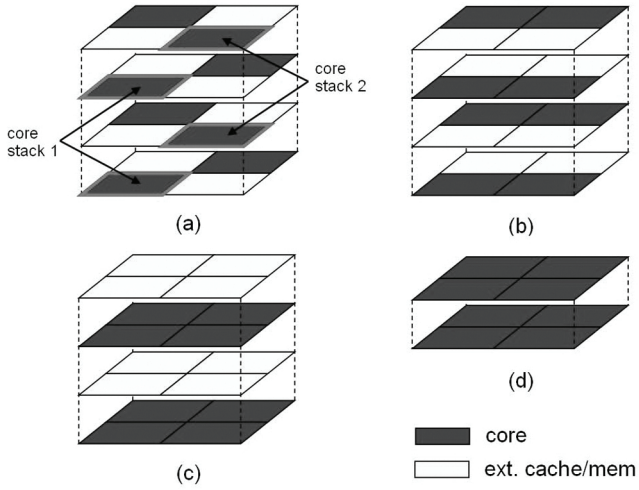


Figure 16: 3D chip multiprocessor floorplan options.

Figure 16(a) and (b) represent the first category in which the distance of some core stacks, e.g., core stack 1 in (a), to the heatsink is different from others such as core stack 2. These floorplans are *thermally heterogeneous*, meaning that the heat dissipation property of different core stacks is different. For example, if the heatsink is on the bottom of the stacked

chip (as illustrated in Figure 17), core stack 2 is further away from the heat sink than core stack 1. Thus, heat dissipation for cores in stack 2 will be more difficult than those in stack 1. In contrast, Figure 16(c) has a rather homogeneous thermal property because all cores are equally distant from the heat sink. Our preliminary work [79] focused only on homogeneous floorplans while this thesis considers both.

Despite these distinctions among different floorplans, they still share some important property. The heat from any core can quickly propagate vertically to other cores above and below. For all these floorplans, the cache layers almost serve as a heat conductance medium between the core layers. Considering this commonality among various 3D floorplans, we choose to use the floorplan in Figure 16(d) as a representative to first introduce the general rationale behind our scheduling algorithms. Then, we will discuss details of our algorithms for homogeneous and heterogeneous floorplans respectively. In Figure 16(d), there are two layers, and each layer contains four cores. The cache banks are subsumed within each core.

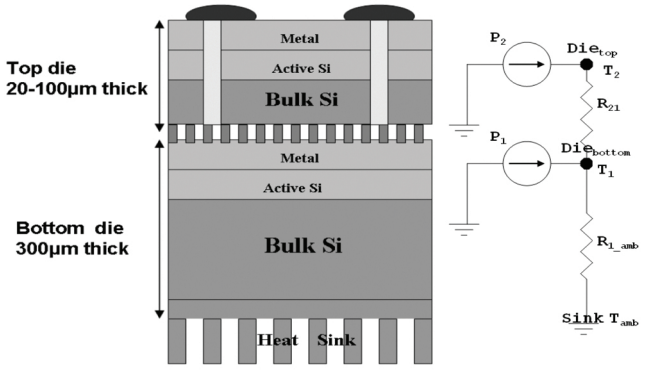


Figure 17: A face-to-back 3D die stacking structure as an example, and the corresponding thermal model.

4.2.1.2 Vertically adjacent layers have strong thermal correlations Similar to a regular 2D processor where heat dissipates mostly in the vertical direction [35], 3D chips also have better heat conductivity in vertical than horizontal direction. This implies that vertically adjacent cores have larger thermal impact among each other than horizontally

adjacent cores. We will use a simple heat transfer model to capture this phenomenon. Figure 17 shows a basic two-layer 3D chip structure (adapted from [7]). We use a face-to-back bonding technology for better scalability in layer count. The top layer is thinned for better electrical characteristics and improved physical construction of the through silicon vias for power delivery and I/O. A thin die also has better heat conductivity than a thick die such as the bottom die. As we can see, the distance between the two active silicon dies are very small ($< 100\mu m$). This directly determines the high heat conductivity between the two adjacent dies. The heat transfer model for this 3D chip is shown on the right of the figure. Here one die is modeled using one node. Its temperature and power are denoted as T and P respectively. R_{21} represents the thermal resistance between the two nodes. R_{1_amb} represents the thermal resistance between the bottom node and the ambient air. We omit the thermal capacitance here to model only the steady state temperature (In our experiments later, both thermal resistance and capacitance are modeled.). Let T_1 and T_2 be the temperature (relative to the ambient air) in the bottom and top node respectively. Then,

$$T_1 = R_{1\_amb}(P_1 + P_2) \tag{4.7}$$

$$T_2 = R_{1\_amb}(P_1 + P_2) + R_{21}P_2 \tag{4.8}$$

Hence, the temperature difference between the two nodes is $R_{21}P_2$. From the parameter used in literature [7, 16, 35, 62], R_{21} is $0.0108 - 0.0159K/W$. P_2 represents the power generated by the entire die. This value is in the range of $40 - 70W$ for a typical single-core processor. Therefore, the temperature difference between the top and bottom die is merely a $0.43 - 1.11K$.

Such a strong thermal correlation between the two adjacent dies can also be demonstrated from our simulation. Figure 18 shows a typical thermal profile of running eight threads concurrently on eight cores as floorplaned in Figure 16(d) (the experimental setup will be introduced in Section 4.2.3). Here eight threads are eight different benchmarks chosen from the benchmark suite we use. We refer to vertically aligned core pairs as a core stack. We

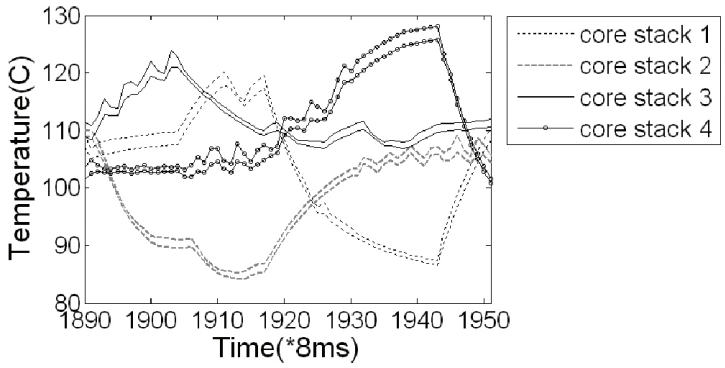


Figure 18: Thermal correlation between adjacent dies.

can see from Figure 18 that there are four distinct clusters of temperature curves. Each cluster has drastically different variations from others. However, each cluster has two lines that are very close to each other. Their variations are almost always synchronized. The four clusters correspond to the four core stacks in the floorplan. And the two lines in each cluster correspond to the temperature variation of the two cores per stack. This experiment shows clearly the strong correlation between adjacent dies, as the temperatures for different core stacks hardly have any dependencies among them, but within each core stack, the temperatures of the two cores are strongly correlated. Such correlation can still be observed for a 4-layer floorplan in our experiments, as the intermediate thin cache layers serve as good heat conductors among their vertical core neighbors.

4.2.1.3 The die layers further from the heat sink are usually hotter Not only are the cores in a stack strongly correlated in their temperatures, but also the ones on the top are usually hotter than those near the bottom. This has also been noted in the literature for steady state temperatures [2, 48]. For clarity, we refer to the cores further from the heat sink as "top" cores, as illustrated in Figure 17. The intuition is that the bottom cores are closer to the heat sink, therefore, their heat can be removed more quickly. Here we give a

more analytical analysis taking into account the thermal capacitance as well. Suppose in the thermal model depicted in Figure 17, the thermal capacitance between the top die and ambient air is C_2. Then,

$$\frac{T_2 - T_1}{R_{21}} = P_2 - C_2 \frac{dT_2}{dt}, \tag{4.9}$$

As mentioned earlier, P_2, which represents the power of a modern processor, has a typical value range of $40 - 70W$. C_2 represents how quickly temperature changes from the top die. For a thin die within $100\mu m$ in a 2-layer 3D chip, the thermal capacitance is reported as $23.6 - 37.4mW \cdot s/K$ [7,62]. dT_2/dt is the temperature change rate within a short time. From our experimental experience, and many other results in the literature, temperature varies slowly with time. For example, we observed a less than $6°C$ increase in temperature in a $8ms$ window using Hotspot 3.0.2 for 3D chips. Hence, the right hand side of equation 4.9 is usually positive with a range of $12 - 52.3W$. Therefore, T_2 is usually higher than T_1.

We also performed simulations to test the above observation. We intentionally put the coolest job (lowest average temperature in a 2D chip) in our benchmark suite on the top die, and the hottest job on the bottom die in a 2-core stacked 3D chip setting. The temperatures of the two cores are shown in Figure 19. We can see that the top core almost always has higher temperatures than the bottom layer. Such an observation serves as a guideline to the development of our heuristic scheduling algorithm.

4.2.2 Scheduling algorithms

The strong correlations among the cores in one stack leads to a scheduling method that considers the entire stack as a whole. The fact that top cores are hotter than the bottom cores suggests that threads within a core stack should be placed with care. Furthermore, we take advantage of this observation and introduce a new voltage/frequency scaling mechanism that results in the fastest temperature drop within the shortest amount of time, once the peak temperature within a stack reaches the thermal threshold. In this section, we present a sequence of thread scheduling algorithms.

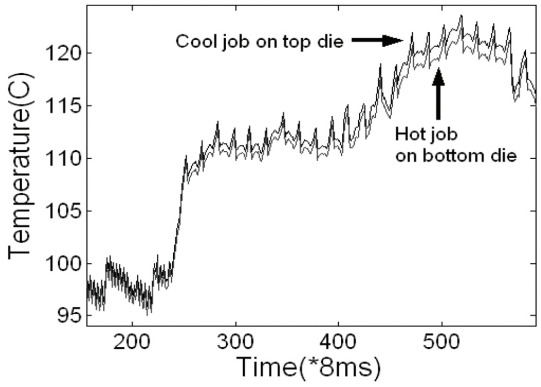


Figure 19: Demonstration of the top die being hotter than the bottom die.

Since we have two categories of floorplans, we will select a representative homogeneous floorplan as shown in Figure 16(c), and a representative heterogeneous floorplan as shown in Figure 16(a). Both the homogeneous and heterogeneous floorplan will be applied with five algorithms: Baseline, Random, Round-robin, Balancing-by-core, and our proposed Balancing-by-stack algorithm.

4.2.2.1 The baseline We use the Linux 2.6 scheduler [9] as our baseline algorithm. In this scheduler, each core has a task queue that keeps track of all running tasks on that core. Each queue contains two priority lists: an active and an expired list. At runtime, the core selects to execute the tasks in the active list, according to some policy. Once a task uses up its time quota, it is moved to the expired list. If all tasks are in the expired list, an epoch has finished, and the scheduler iterates the process by swapping the two lists. Each task in the active list has $10 - 200ms$ of CPU cycle quota, depending on its own priority. By default, the core switches to a different task every $100ms$. Thus, in our 8-core 3D chip, upon the scheduling interval of every $100ms$, the scheduler selects a task from each core's active list according to its original policy, and then assigns it to a different randomly selected core.

This algorithm is simple, and has low context switch overhead compared to other algorithms introduced later. However, it may run the risk of putting two hot tasks into the same core stack, which may lead to an extremely high temperature that results in a long and harsh voltage/frequency scaling penalty for both tasks. Moreover, once a poor scheduling has been made, it stays in that condition for a long period of time ($100ms$ until the next scheduling time), exacerbating the already serious thermal condition within the chip.

4.2.2.2 Random (Baseline+) A quick fix of the baseline scheduler is to increase the scheduling frequency. In the normal Linux OS, any context switch interval between $10 - 200ms$ may be used [9]. A minimum of $10ms$ is recommended to avoid unnecessary context switch overhead. We used $8ms$ as our scheduling interval mainly due to an experimental restriction on collecting the power traces. Also, $8ms$ is close to the thermal constant of the core under test. However, the algorithm can be directly applied to any scheduling interval recommended in Linux such as $10ms$ if those restrictions do not apply. Further, we take into account the extra context switch overhead using an $8ms$ scheduling interval during our experiments. We performed a real machine measurement on the time required to perform a single context switch. For an $8ms$ interval, it is $\sim 0.44\%$, a mild penalty that can be easily offset by the performance gain from a better scheduling method.

With the improved baseline scheduling algorithm (termed Random to reflect the scheduling decision), the chip can exit a poor thermal condition due to an unwise scheduling more quickly, resulting in less harmful impact.

4.2.2.3 Round-Robin The Random scheduler may result in uneven distribution of power and temperature as tasks are assigned randomly to any core. A Round-Robin scheduler (RR) can overcome this by rotating tasks among cores in a fixed order periodically. Therefore, after N iterations where N is the number of cores, each task has executed on every core for one scheduling interval, e.g., $8ms$. This can help balance the power and temperature distribution in the long run.

4.2.2.4 Temperature balancing by core An alternative way to balance the heat among the cores is to explicitly arrange the tasks according to their power consumption and the core temperatures. Essentially, a high power task should be assigned to a low temperature core. At each scheduling point, the scheduler sorts the power consumption of all tasks and the current temperature of each core. It then assigns the task with the highest power to the coolest core, the 2^{nd} highest power to the 2^{nd} coolest core, and so forth.

Such a mechanism should perform a better job in balancing the temperature distribution among cores than RR. However, recall that there is a strong thermal correlation between two adjacent layers, and the cores in one stack have only a small difference in temperatures. This implies that if a core stack contains the hottest core, it probably also contains the 2^{nd} hottest core. When the temperature Balancing-by-core algorithm is applied, the tasks with the lowest and 2^{nd} lowest power are scheduled to this hot core stack. Similarly, the tasks with the highest and 2^{nd} highest power will be scheduled to the coolest core stack. After that, the hottest/coolest core stack will have the largest temperature drop/rise, which may lead to temperature oscillations and task thrashing between those two stacks, potentially leading to more thermal emergencies. In that case, a RR, or a Random algorithm may be a better solution.

Another issue with this mechanism is how the power consumption of each task is obtained. Recently, there has been proposals on obtaining the runtime power consumption of an application through probing the *performance counters* in a processor [37]. We also adopt this approach and assume that each core is equipped with such counters that can be used for power estimation. Note that our power estimation need not be very accurate, as we only need the sorted order of the power, not the absolute values.

4.2.2.5 Temperature balancing by stack The core-based temperature balancing algorithm can create thrashing of tasks between the hottest core stack and the coolest core stack, as we analyzed earlier. This is because the algorithm, while trying to balance the temperatures among all cores, treats each core independently. However, as adjacent dies

have strong temperature correlations, cores in the same stack should indeed be considered together. Intuitively, we can assume that each stack is a "super" core that has cores with similar temperatures. Hence, scheduling of the tasks within three dimensions can be reduced to scheduling of "super" tasks within two dimensions. Here a super task is defined as a set of tasks that are assigned to a super core, i.e., a core stack.

We treat homogeneous and heterogeneous floorplans differently in this algorithm as their super cores have different thermal properties. We first elaborate on the algorithm for the homogeneous floorplan.

Super tasks. Let L be the number of layers in a 3D chip, and N be the number of cores per layer. As a super core contains L cores, a super task should also contain L tasks and there are N super tasks. The scheduling of N super tasks among N super cores is now simply a 2D problem, where a balanced temperature distribution is desired. Hence, we first balance the power among super tasks, i.e., let each super task have about the same power, and then balance the temperatures among super cores by scheduling a relatively high power super task onto a relatively cool super core.

To balance the power among super tasks, we first sort the powers of all $N \times L$ tasks. Let B_{1-N} be N initially empty bins. We will fill powers into these bins such that each bin will contain L tasks, and the total powers of each bin are about the same. In descending order of powers, we put each power value into a bin that has the smallest current total power among all bins. This policy attempts to reduce the gap between the smallest and the largest total power in each step, in order to generate a relatively balanced total power across N bins. Finally, all powers within a bin form a super task. We remark that our policy is only a heuristic as an optimum solution may require an exhaustive search. We aim for a simple, effective, yet low-complexity heuristic because the scheduler makes the decision at runtime. Here we point out this problem is a NP-complete problem, but our solution belongs to one of the good heuristics used to solve the partition problem.

Task distribution among and within super cores. The goal of producing super tasks is to generate relatively balanced power distribution across super cores. Once the super tasks are formed, we sum up the temperatures of all L cores in a super core, and sort them. Similar to the previous procedure, we assign the hottest super core with the super task of the lowest power, and so on. Figure 20 shows an example of scheduling 8 tasks onto a 2-layer, 4-core-per-layer, 3D chip. Step (a)-(c) depict the procedure except for how tasks within a super task are allocated onto different cores within a stack.

As discussed earlier, the top cores are usually hotter than the bottom cores in a core stack. Hence, we should allocate tasks of higher powers onto the bottom cores for better heat removal, and tasks of lower powers onto the top cores. For example, if the temperatures of the cores from bottom up are strictly increasing, then the tasks allocated to them should have strictly decreasing powers from bottom up. Figure 20's last step illustrates this policy in a two-layer floorplan.

Scheduling procedure. To summarize, on every scheduling interval ($8ms$ in our case), the scheduler performs the following steps:

1. Sort the powers of all tasks. Form super tasks. Sort the power sums of the super tasks from *low to high*.
2. For each super core, sum up the temperatures for all cores. Sort the temperature sums for all super cores from *high to low*.
3. Create a sequential one-one mapping between the sorted super tasks and sorted super cores.
4. In each super core, allocate the tasks in their increasing power order onto the cores with decreasing temperature order.

Our algorithm involves mostly sorting of the powers and temperatures. Therefore, its time complexity is $O(NL \log (NL))$.

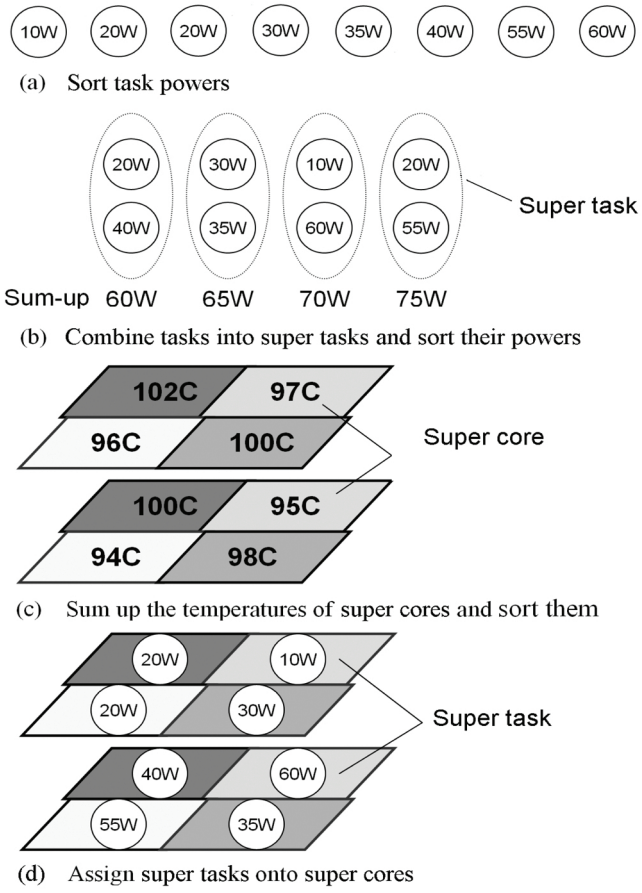


Figure 20: The temperature balancing-by-stack algorithm.

The major difference in heterogeneous floorplans is that different super cores have different heat dissipation capability due to their varying distances to the heatsink. For this reason, even if two super cores are of the same present temperature, the same super task assigned to them will result in different future temperatures. For example, in our experiment for the floorplan shown in Figure 16 (a), we assigned eight identical tasks onto eight cores and still observed 4-7K thermal difference on the top four cores. Therefore, unlike the algorithm for homogeneous floorplans where the total power among super tasks should be well balanced, the task bundling in the heterogeneous floorplan should intentionally create a power imbalance to generate a balanced temperature distribution among super cores. However, it is difficult to estimate how much power difference we should create among super tasks because the future temperature depends on not only power but also the present temperature and duration of the power. Therefore, for a given set of power values, our algorithm forms the super tasks with minimum, moderate and maximum total power difference (denoted as Min-diff, Mod-diff and Max-diff respectively), and dynamically make the selection of super tasks.

Let $P_1 \cdots P_n$ be n powers in ascending order. Super tasks with Min-diff, Mod-diff and Max-diff are formed as follows, assuming each super task contains L tasks:

- Max_diff: $\{P_1, P_2 \cdots P_L\}, \{P_{L+1}, \cdots P_{2L}\}, \cdots$
- Mod_diff: $\{P_1, P_{L+1}, P_{2L+1} \cdots\}, \{P_2, P_{L+2}, \cdots\}, \cdots$
- Min_diff: The principle is to balance the total powers of super tasks. This is identical to the algorithm for homogeneous floorplans (Section 4.2.2.5).

Intuitively, when the temperature difference among super cores is large, a super task with Max-diff is desired. However, if the power difference among tasks is also large, using the Max-diff may be an overkill. A Mod-diff combination may be sufficient. Therefore, our decision relies on both the temperature gradient (denoted as ΔT) among the super cores and the power range (denoted as ΔP) of the tasks. Let

$$\theta = \frac{\Delta T}{\Delta P}. \tag{4.10}$$

When θ is small, the temperature gradient (ΔT) is relatively small compared with the power range (ΔP) of the tasks. Super tasks of Min-diff are more appropriate in this situation because we need only to perform mild temperature adjustment. On the other hand, when θ is large, a more aggressive task bundling to create power differences is necessary, hence the selection will favor Max-diff.

During our experiments, we use two heuristic θ values: $\theta_1 = 0.5$ and $\theta_2 = 1$ as the thresholds for choosing different algorithms. The choice of these two values is based on our experimental settings, and may vary with thermal properties of the floorplan. If θ falls in the range of $[0, \theta_1)$, Min_diff will be chosen. If θ is in the range of $[\theta_1, \theta_2]$, Mod_diff is selected. If θ is greater than θ_2, Max_diff will be selected. Furthermore, if ΔT is really very small(in our case it needs to be less than $0.8°C$), this indicates the current task combination and assignment is working well. In this situation, the tasks stick to the cores for the next scheduling interval.

A critical component in concert with our proposed scheduling algorithm is how to handle thermal emergencies once a core temperature increases above the hardware threshold. Conventionally, such a core will be put into a low power state through DVFS. In a 3D chip, since the top cores are usually hotter, thermal emergencies usually occur in the top layers. Moreover, our scheduler puts cooler tasks on the top layers, which means that those tasks are more likely to undergo DVFS, leading to their degraded performance.

The problems of such conventional thermal management are twofold. First, the cooler tasks could be penalized more often than the hotter tasks, which brings a fairness issue among different tasks. Intuitively, hotter tasks should be restrained by the system due to their potential harmful impact on the chip. Second, applying DVFS to the cooler tasks on the top layers does not yield the same efficiency as in a 2D chip. This is because it takes a longer time to cool down the top cores due to their high power neighbors at the bottom. In fact, it is because of those hot bottom tasks that the top cores are overly heated. Therefore, a more rational thermal management should employ the scalings to the source of the overheating — the bottom cores that are running high power tasks.

More formally, when core A of a super core S is overheated, the thermal management will select core B with the highest power in S to engage DVFS. B may or may not be identical to A. Such a thermal management strategy solves the two problems above effectively. First, cool tasks are not penalized more often than hot tasks because if a cool task becomes a temperature victim, the hot task that caused the problem is penalized. Second, all cores in S, including A and B, are quickly cooled because the total power of S is reduced with the maximum strength. For example in Figure 20, if the super core containing the 20W-40W super task tripped a thermal emergency on the 20W core, and suppose the DFVS reduces the power of a core by half, then our scheme will reduce the total power of this super core to $20 + 40/2 = 40W$, while the conventional thermal management will only reduce it to $20/2 + 40 = 50W$. As we can see, if DVFS is applied to a relatively low power task, the result is inferior because a task is being penalized, but the total power in the chip is not reduced as much. This is often the case for the temperature Balancing-by-core scheduler as it tends to allocate cool tasks on the top layer (since it is usually hotter).

As a result, our mechanism brings down the temperature of the hotspot at the highest speed, resulting in minimum penalty to the overall performance of this super core. We will show later that our proposed temperature Balancing-by-stack scheduling algorithm with improved thermal management results in many fewer thermal emergencies and the much better performance among all previous schemes.

4.2.3 Experimental methodology

4.2.3.1 Floorplan setup Our detailed experiments are conducted on floorplans as depicted in Figure 16(a) and (c). Each floorplan has four layers and a total of eight cores. We simulated 8 P4 Northwood cores at 3.0GHz clock frequency. Each core has a size of $1.144 \times 1.144\ cm^2$. The remaining space is left for extended cache or memory. The total die size is $2.289 \times 2.289 cm^2$. Other physical parameters such as layer thickness and thermal conductivity of Cu and Si are adopted from [7]. For example, the top three layers are thinned to $20\mu m$ while the bulk Si layer closest to the heat sink is of several hundreds of μm.

4.2.3.2 Simulation tool and power trace collection

We used Hotspot [35] version 3.0.2 as our simulation tool. We chose the grid model to experiment with our 3D floorplan. We substituted the 4th-order Runge-Kutta method with TILTS [26] to generate accurate temperatures at high speed.

Hotspot takes power traces as inputs, and temperature variation within a die is a slower process compared to other metrics such as IPC. Hence, we need to collect extended power traces to model realistic temperature variations such as warming up and cooling down due to task scheduling. As mentioned earlier, we adopt the recently proposed performance counter based method [37, 73] to collect runtime hardware activities of a program on a real machine. We obtained the power model (calibrated) from [37, 73] to produce long power traces for programs from a Linux machine with a Pentium 4 core. The traces contain powers for each functional unit, and all traces are a complete execution of the programs in SPEC2K.

For scheduling algorithms that require power information (Balancing-by-core and Balancing-by-stack), we use the power in the last $8ms$ interval to predict the power in the next interval. That is, the scheduling decisions are based on local power information. The scheduler does not need to know whether a program is globally hot or cool. Also, we use the last power predictor in the scheduler due to its simplicity. We experimented with more complex power predictors and found that their overhead, both in time and space, is not appropriate for on-line scheduling [76]. Most of the benchmarks exhibit $\sim 5\%$ power mis-prediction rate. Our experiments show that an error within 5% makes last power prediction accurate enough for the scheduler.

4.2.3.3 Benchmark classification

We first ran the power traces of each benchmark to obtain its temperature profile as shown in Figure 21. The power traces ran in the HotSpot thermal simulator. Because the RC constant of the simulator is a little different from Pentium 4 processor, this thermal profile is a little different from Figure 6 in the previous section. We next classified these benchmarks as hot(power-intensive), cool(power non-intensive), and mild(between hot and cool). After that, we created 9 workload combinations, as listed

in Table 3, each with one or more hot tasks. The workload mixes without hot tasks are less thermally critical and thus, are not considered here. In Table 3, when the number of benchmarks in one combination is less than 8, copies of the benchmarks were created to ensure that every core in the floorplan has one task to run. This resembles the situation of running parallel threads of the same program in multicore processors.

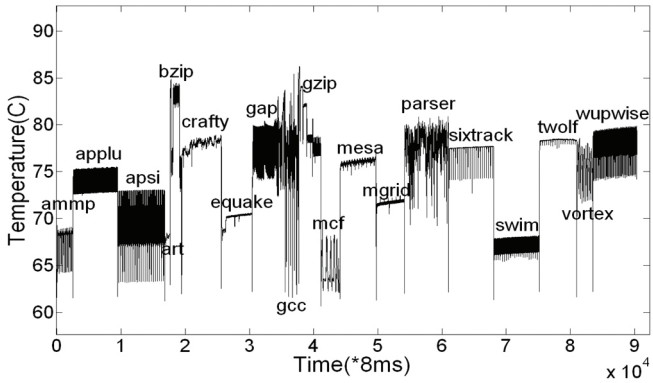


Figure 21: Temperatures of the benchmark in SPEC2000

4.2.3.4 DVFS implementation and context switching overhead We modified Hotspot to incorporate the hardware DVFS. Every 80μs, 1/100 of a scheduling interval, Hotspot checks if the temperature has trespassed the threshold. If so, the voltage is lowered from $1.3V$ to $1.1V$ and the frequency is reduced by 4/5. We charge 30 μs of overhead on every voltage/frequency transition. During a DVFS scaling, if the temperature persists above the threshold after one 80μs, the scaling continues and no additional DVFS switch overhead is charged. We do not choose multi-level DVFS scheme to avoid unnecessary switch overhead in every level transition.

The other overhead in our proposed scheduler is the increased number of context switches. We measured this time in a Linux machine by enforcing a large number of context switches between two tasks, and calculating the average switch time from the increased execution time of these two tasks. Such measurement also includes the cache warm-up time required by the

Table 3: The combination of benchmarks in simulation

HC	crafty mcf
HC	sixtrack swim
HHCC	bzip twolf art ammp
HMMC	wupwise equake applu ammp
HM	gzip mgrid
HM	parser equake
HHMM	crafty gzip mgrid apsi
HHMMMCCC	gap twolf equake mgrid vortex ammp art swim
HHHHCCCC	bzip gzip sixtrack wupwise ammp art mcf swim

tasks. This quantity in our test machine is $\sim 35\mu s$. Later we will see that our proposed scheduler can still outperform Linux baseline scheduler even with much higher context switch frequency.

We set the thermal threshold to trigger DVFS as $105°C$. This threshold introduces 6%∼25% (12.4% on average) thermal emergencies in the task mix, which account for 4%∼16% (8.4% on average) performance degradation under Linux baseline algorithm. Note that the thermal intensity of applications is a feature relative to the emergency threshold. For example, if the average temperature is close/far to/from the threshold, then this application is considered hot/cool. Hence, testing a high threshold would make most programs "cool", and scheduling cool threads is not necessary. Testing on an overly low threshold would make most programs "hot", which is unrealistic and scheduling would not help anyway. Therefore, we chose 105°C to present practical scenarios and to give reasonable room for scheduling threads of different thermal intensity.

4.2.4 Results and analysis

The metrics we use to evaluate different scheduling algorithms are peak temperature of all cores, the reduction in time that a task stays above the thermal threshold (termed "thermal emergency reduction" in later discussion), and performance improvement in terms of total execution time reduction of all tasks. The peak temperature indicates how well a scheduler can alleviate the worst cases of the thermal condition on-chip. The thermal emergency reduction indicates the capability of a scheduler to control the temperature below the hardware threshold. The performance improvement is the result of both the thermal emergency reduction and the efficiency of lowering the temperature during an emergency. Next, we present the results for homogeneous and heterogeneous floorplans separately.

4.2.4.1 Homogeneous floorplan In the following we will introduce the experiment results on the thermally homogeneous floorplan. Five schedulers, Baseline, Random, Roundrobin, Balancing-by-core, and Balancing-by-stack, are tested in the experiments.

First, let us see a qualitative comparison among different schedulers on the homogeneous floorplan. Figure 22 shows a close-up of temperature traces for 8 cores running the HMMC workload under different scheduling algorithms. Here, we did not enforce DVFS at the threshold because otherwise, many high temperature curves would be capped at the threshold. As we can see, the baseline algorithm can result in a large temperature gradient across different core stacks. A $\sim 34°C$ difference between the hottest and the coolest core stack is observed in this figure. For Random and RR scheduler, the temperature gradient within the 3D chip gradually reduces because their scheduling interval is $8ms$, much smaller than that in the baseline. The temperature gradient is between 4-19$°C$ in these schedulers. Finally, both the Balancing-by-core and our proposed Balancing-by-stack schedulers create the smallest temperature gradient among all cores. The temperature curves of all cores almost overlap entirely. The width of the temperature band is 2-6$°C$ only, indicating an excellent balance of temperature among the cores. However, the Balancing-by-core scheduler generates more fluctuation. Note that an ideal temperature balancer would create a $0°C$ difference among all cores. Hence, our proposed Balancing-by-stack algorithm is only a couple of degrees from the ideal case.

Balancing the temperatures across the chip can help to reduce the peak temperatures among all cores. Figure 23 shows the peak temperature generated from each scheduling algorithm assuming there are no DVFS employed (otherwise, the peak temperature is just the thermal threshold). We can see from the figures that the baseline algorithm can generate the highest peak temperature of $118.31°C$. The Random, RR, Balancing-by-core and Balancing-by-stack can reduce the peak temperature better and better. Our proposed Balancing-by-stack scheduling generates the second lowest peak temperature of $113.71°C$, $4.6°C$ lower than the baseline and a mere $0.03°C$ higher than that of Balancing-by-core.

A direct benefit from scheduling the tasks is the reduced thermal emergency time, i.e. the time a core temperature stays above the hardware thermal threshold. Note that this metric does not necessarily correlate with the peak temperatures reported in Figure 23, which are collected under *no DVFS*. For example, a relatively low peak temperature may

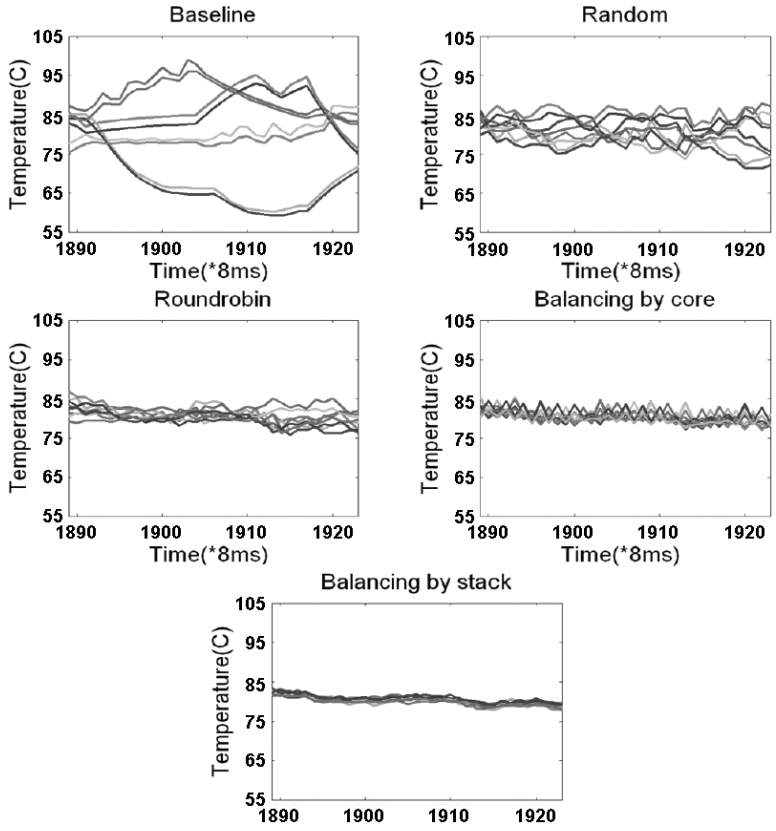


Figure 22: A zoom-in of temperature variation over time under different scheduling algorithms.

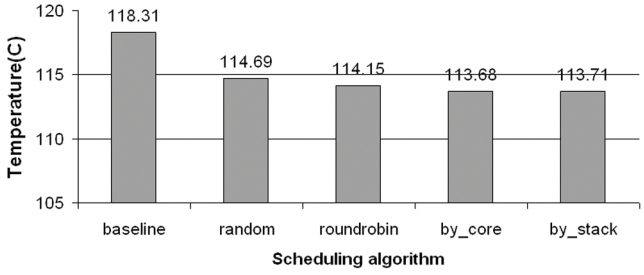


Figure 23: Peak temperatures of different scheduling algorithms.

still trip DVFS if temperature oscillates around the threshold often. Figure 24 shows thermal emergency time reductions from different algorithms, normalized to the baseline case. As we can see, the Random, RR and Balancing-by-core can reduce the emergency time by 30.9%, 37.41% and 36.4% on average respectively. Our Balancing-by-stack algorithm removes the most emergency time in 8 cases of 9 benchmarks. An average of 46.23% reduction is observed, with a range of 6.06%-96.04%. Also, the Balancing-by-core algorithm turns out to introduce as much emergency time as RR algorithms even with lower peak temperature. This is because (1) it tends to create temperature oscillations among core stacks as discussed in Section 4.2.2.4; and (2) it tends to allocate cooler tasks on the top layer where DVFS is usually engaged for a long time. The consequence is that the overall power in the entire chip is not reduced as much as in other schedulers, where high power tasks can be scaled during emergencies. Therefore, a Balancing-by-core scheduler may not be a good scheduling candidate in practice.

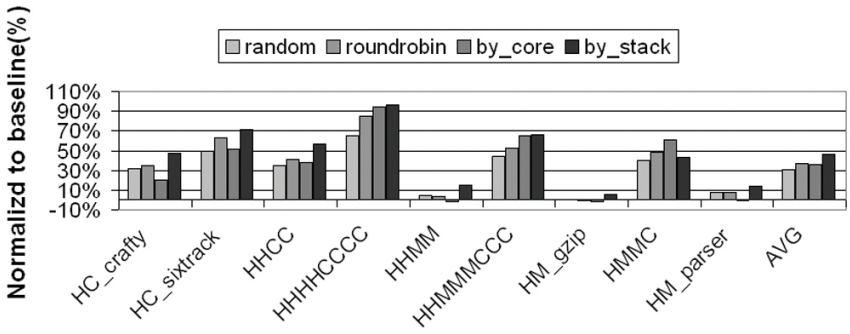


Figure 24: Thermal emergency time reductions in homogeneous floorplans.

Corresponding to the thermal emergencies removed, our proposed Balancing-by-stack algorithm achieves the best performance improvement among all algorithms discussed. This is shown in Figure 25. The performance is the total execution time of all 8 tasks in a workload. The results are normalized to the baseline performance. On average, the Balancing-by-stack achieves a 5.11% improvement, while the Random, RR, and Balancing-by-core algorithm achieve 1.45%, 1.72% and 1.65% improvement respectively. This is primarily due to the

number of thermal emergencies our algorithm removed, as well as the high efficiency in handling them with the new thermal management mechanism proposed by us.

We also notice that for some occasions, the performance may not improve even if the thermal emergency time is reduced. This could happen when the temperature floats around the thermal threshold, but does not increase overly high. In such a scenario, there could be many DVFS triggered, which introduce a high transition penalty and overkills the gains from scheduling. For example in the HMMC workload, the Balancing-by-core removed 18.01% of thermal emergency time in the Balancing-by-stack, but its performance is 0.54% worse than the Balancing-by-stack. Our Balancing-by-stack removes more thermal emergency time than other schedulers in 8 cases of 9 benchmark combinations, and therefore, achieves the most performance improvement.

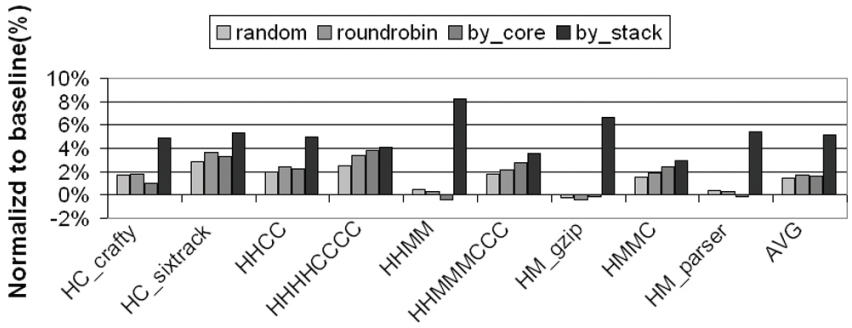


Figure 25: Performance improvements for homogeneous floorplans.

Since Balancing-by-stack utilizes three heuristics, we want to investigate the individual contribution of each heuristic. We call the heuristic to always trigger DVFS on the most power-intensive job in the same stack as *powsca*, and call the heuristic to move the hotter jobs closer to the heat sink as *hotseq*. Finally we call the heuristic to balance the power among core stacks as *balance*. We conducted the experiment and the results are shown in Figure 26. All the results are relative to the performance improvement of the Random scheduler. The first bar in each group shows the performance improvement compared to the Random scheduler when only using *powsca*. On the average, this heuristic achieves

1.71% improvement. The third bar shows that using *balance* gets only 0.39% improvement on the average. In particular, using *balance* is worse than the Random scheduler under the workloads such as HHMM, HM_gzip and HM_parser. The reason is that *balance* tends to make the thermal traces smooth and avoid thermal peaks, seen in Figure 22. When the thermal traces are all very close to the threshold, averaging power among the core stacks makes each core stack trigger lots of DVFS. On the contrary, Random can heat up some cores and leave some other cores cool, as shown in Figure 22. Thus at least some cores do not trigger DVFS. *hotseq* itself can not improve performance, because it always assigns the hot jobs onto the cores close to the heat sink. When DVFS is triggered, it's the cool jobs away from the sink getting penalized. So the total amount of power in one stack does not decrease significantly. However, *hotseq* can aid *powsca* and *balance* to improve the performance by 3.15% and 1.63% respectively.

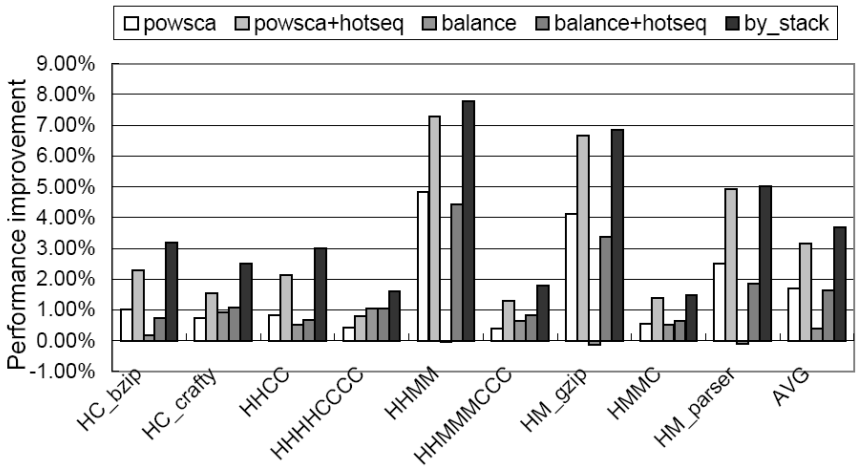


Figure 26: The individual and combined effects of three heuristics. The results are relative to that of the Random scheduler.

4.2.4.2 Heterogeneous floorplan In addition to the five algorithms applied to the homogeneous floorplan, two additional algorithms are also tested for heterogeneous floorplans. The first is the revised Balancing-by-stack algorithm with dynamic super task forming mech-

anisms. The algorithm is designed to tackle the thermal heterogeneity of the floorplan as discussed in Section 4.2.2.5. The second is a pseudo-optimal algorithm that tests the quality of each algorithm discussed. We term this algorithm a "1-step-optimal" since it tries all task bundling mechanisms and chooses the one that triggers the fewest DTMs in *one next step*. Notice that this is not a true optimal algorithm which would go beyond one-step to enumerate all possible schedules and pick the optimum one (and so is termed "1-step" only). Although it is not realistic to adopt "1-step-optimal" algorithm online due to its complexity, it does indicate the potential for improvement of the discussed algorithms.

Figure 27 shows the thermal emergency time reduction for different algorithms normalized to the baseline case. As we can see, Random and RR perform relatively poorly compared with other algorithms because of the heterogeneity in the floorplan. They achieve 12.41% and 12.35% of thermal emergency time reduction respectively. Our proposed dynamic Balancing-by-stack algorithm achieves a total of 46.37% reduction, only 1.92% away from the 1-step-optimal on average, and is better than the remaining algorithms. For example, it removes 9.22% more emergency time than the original Balancing-by-stack algorithm. This indicates that dynamically tuning of the task bundling is very helpful to a thermally heterogeneous floorplan. The Balancing-by-core algorithm is slightly better than dynamic Balancing-by-stack in three cases: HHHHCCCC, HHMMMCCC and HM_gzip. This is because when ΔT and ΔP do not change, our dynamic Balancing-by-stack algorithm will select the same one from the fixed power bundling schemes, while a slight re-ordering of core temperature will cause Balancing-by-core to form a different and better power bundle more flexibly. Also, Balancing-by-core slightly surpasses 1-step-optimal in HHMMMCCC and HM_gzip workloads. This is because the 1-step-optimal does not generate a global optimal schedule.

Compared with the thermal emergencies removed, our dynamic Balancing-by-stack algorithm achieves the best performance improvements on average among all the algorithms except 1-step-optimal.

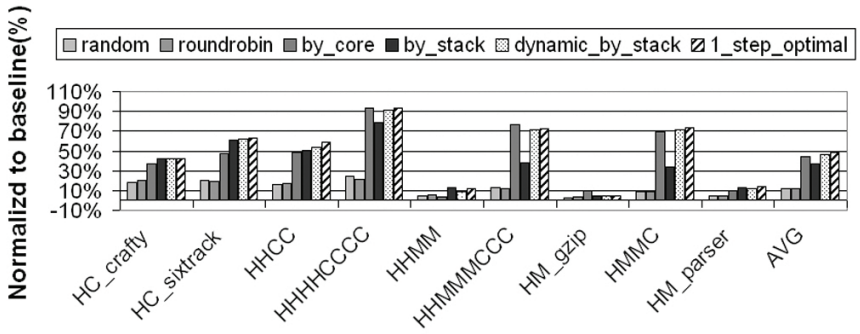


Figure 27: Thermal emergency time reductions in heterogeneous floorplans.

Figure 28 shows that Random and RR achieve 0.39% and 0.31% improvement respectively, which is notably lower than 1.45% and 1.72% improvement shown in Figure 25, indicating that Random and RR are not as helpful in heterogeneous floorplans as in homogeneous ones. Balancing-by-stack achieves 2.46% improvement more than Balancing-by-core, though Figure 27 shows it removes 6.74% less thermal emergency time than Balancing-by-core. The reason behind this is that Balancing-by-core tends to generate a lot of overhead in DTM mode switches though the total *time* above the emergency threshold is low which was reported in Figure 27. Finally, the dynamic Balancing-by-stack algorithm achieves the best performance improvement of 4.78% with negligible gap from the 1-step-optimal.

4.3 MAXIMUM BIPARTITE MATCHING IN CMP WITH PROCESS VARIATION

As IC technology nodes continually scale down to 45nm and below, there is significant within-die process variation in the current and near-future CMPs. Process variation(PV) makes the cores in the chip differ in their maximum operable frequency, and the amount of leakage power they consume. To take advantage of the frequency variation of the cores caused by process

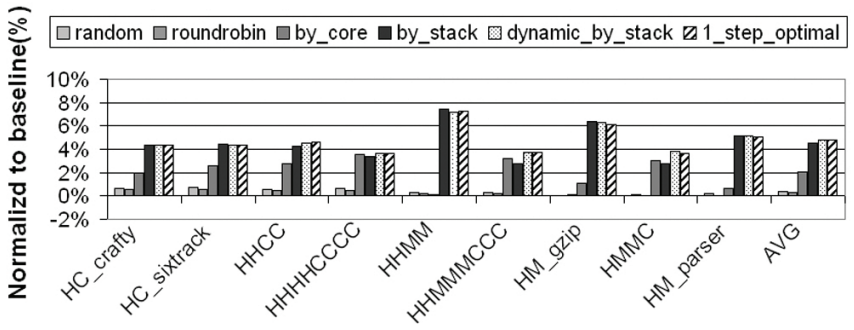


Figure 28: Performance improvements for heterogeneous floorplan.

variation in CMPs, Teodorescu et al. [68] proposed an algorithm named *VarF&AppIPC* to map higher-IPC(instructions per cycle) cores to faster cores in order to obtain higher overall throughput. The reason behind this approach is that low-IPC applications are often memory-bound and usually benefit less from high-frequency cores than high-IPC applications do.

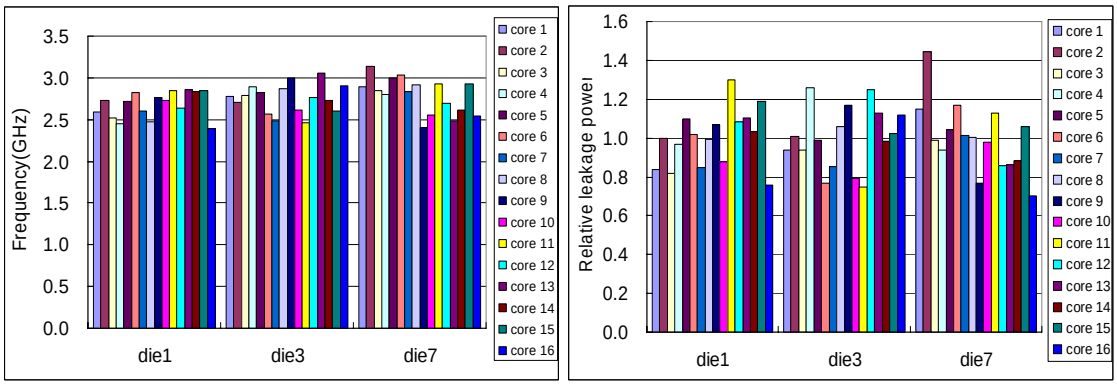
We will demontrate that *VarF&AppIPC* might not be able to achieve as high throughput as it intends to do under thermal constraints. We propose here a task migration algorithm that tries to maximize throughput by taking both thermal and PV issues into account. The algorithm not only considers the frequency and leakage power information on each core, but also considers the power characteristics of running jobs (tasks). With that information, the algorithm predicts the throughput of each core-job binding, and uses the Maximum Bipartite Matching algorithm to get the optimal mapping.

4.3.1 Motivation

The within-die process variation has a significant impact on the CMP fabricated with 45nm technology nodes and below. We inverstigated the process variation using PV modeling method described in [59]. The details of the modeling will be introduced in the experiment

setup part in Section 4.3.4.2. Figure 29 shows the discrepancies of the maximum possible frequencies and the leakage power among the cores on our sample 16-core die 1, 3 and 7, which are picked from a total of 600 dies we modeled. The result in the figure demonstrates that running the CMP at the frequency of the slowest core can waste the computational power of the CMP up to 10%, because the higher-frequencies of the faster cores are wasted.

Teodorescu et al. [68] advocate letting the cores run at their maximum frequencies. Furthermore, they always map the higher-IPC jobs onto faster cores, because low-IPC jobs are often memory-bound and benefit less from high-frequency. This algorithm is known as *VarF&AppIPC* in [68]. *VarF* means the frequencies of the cores are varied, and *AppIPC* means the IPC of the jobs are considered in their scheduling algorithm. Later we refer to this algorithm simply as *AppIPC*, because all the algorithms in this work will run on cores that have varied frequencies.



(a) (b)

Figure 29: (a)The variation of the frequencies of the cores on sample die 1, 3 and 7 among the 20 sample dies. (b) The variation of the leakage power of the cores on sample die 1, 3 and 7 at the temperature of 100C.

Although *AppIPC* can produce higher throughput when the die is under no thermal limit, it's actually not effective when there are thermal constraints to be met under thermal management. The binding of a high-IPC job to a fast core means more instructions per

second(IPS), which typically translates to higher dynamic power. Furthurmore, faster cores typicaly contain a large portion of gates with low V_{th}, so the leakage power in the faster cores is often higher. The combined effect of high dynamic and leakage power is that faster cores are more likely to run into thermal emergencies and trigger hardware DVFS. As a result, the throughput on the faster cores will not be as high as it is supposed to be. Therefore, the effectiveness of *AppIPC* is reduced.

To verify this, we make a preliminary study in a 16-core CMP with process variation by running 4 to 16 jobs(tasks). Each bar in Figure 30 actually represents the average results from 8 different workloads running on 20 sample dies. *Fixed* means that the scheduling binds each job on one core from the beginning to the end of the job execution, which mimics the Linux baseline algorithm. *NoDVFS* represents the ideal scenario where there are no thermal constraints so that no DVFS is triggered. In the left of Figure 30, the first two bars in each group shows that *AppIPC_noDVFS* achieves higher throughput over *Fixed_noDVFS* by 0% to 6.7% when the number of jobs n decreases from 16 to 4. The reason why there is barely no performance improvement when $n = 16$ is that *AppIPC* needs to track the IPC of jobs dynamically, and migrates jobs if necessary. Therefore, the overhead of task migration offsets the gain of *AppIPC*. The right two bars in each group show that under thermal constraints and DVFS, *AppIPC_DVFS* achieves -0.8% to 4.7% throughput improvement over *Fixed_DVFS*. The reduction of *AppIPC* effectiveness can be explained by the amount of DVFS triggered. The right of Figure 30 shows *AppIPC* triggers more DVFS than *Fixed* no matter how many jobs there are. For example, when $n = 8$, 12.7% more DVFS is triggered by *AppIPC*. Therefore, when there are thermal constraints, although *AppIPC* can still map higher-IPC jobs to faster cores, it suffers from more of a throughput penalty from DVFS compared to *Fixed*. This reduces the effectiveness of *AppIPC*. To avoid a penalty from DVFS, the best way [22] is to move the high-IPC job to a cool core, which can potentially be slow. In this case, the high-IPC job can not enjoy the relatively high frequency on the faster cores. Therefore there is not a clear view of choosing which core: a slow one, or a fast one that triggers DVFS, and we are motivated to find a good metric to make an effective judgement.

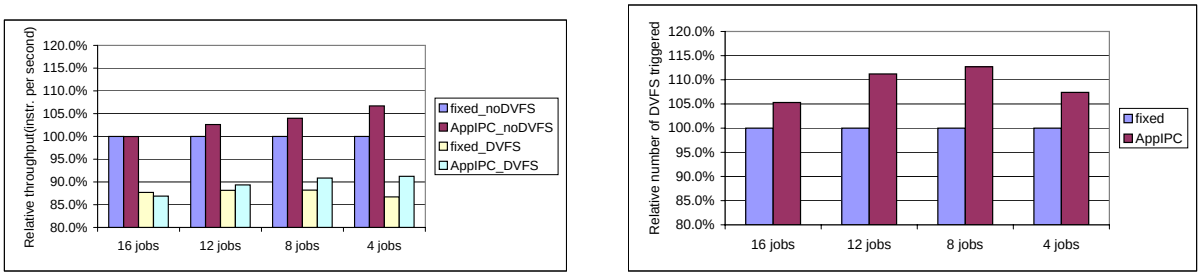


Figure 30: The relative throughput attained(left) and the relative DVFS triggered(right) by running varied number of jobs when the interval is 8ms and the thermal environment is hot.

4.3.2 MBM algorithm

For any mapping of jobs onto cores, there is always a corresponding overall throughput. We believe the the overall throughput on the cores is the ultimate objective of a good mapping. To achieve the optimal overall throughput, simply considering IPC of jobs and frequencies of cores seperately as did in *AppIPC* is not enough.

Imagine there is only one job to be assigned to a n-core CMP in the next scheduling interval, if we can predict the IPS(throughput) of the job when running on the cores, it is not difficult for us to decide which core is better for maximizing the throughput. Under the thermal constraint, the candidate core may not be the fastest core because the fastest core might approach the thermal limit and can trigger a lot of DVFS. It may not be the core as fast as possible but not triggering any DVFS, because the benefit from a much faster core may outweigh the loss brought by a little DVFS on that core. The success of picking the most suitable core relies on the accurate prediction of IPS, which will be introduced in detail in Section 4.3.3. Now, considering a more general case when there are n cores and n jobs, the problem becomes complex: How to choose the best mapping from $n!$ possibilities of 1-1 mapping?

This problem is described in Figure 31 as a weighted complete bipartite graph, in which the jobs and the cores are vertices on the top and at the bottom in the graph. The binding of one job and one core corresponds to one edge with the corresponding IPS as the weight. This problem can be solved by the classical maximum bipartite matching algorithm [72](referred as *MBM* in this thesis). The time complexity of running this algorithm is $O(V^2 \log(V)+VE)$, where V and E is the number of cores and edges respectively. In practice, the computation time is not a big overhead for the current and near-future CMP. When the core number $n = 16$, the computation time to get the optimal result is $16\mu s$ using a 2.8GHz Intel Xeon CPU; when $n = 64$, the time is $58\mu s$. If the scheduling interval length is $100ms$, which is very typical in a current Linux OS [9], the overhead is just 0.058%. Compared with this algorithm, an exhaustive search for the optimal solution will take about 2.6 days when $n = 16$.

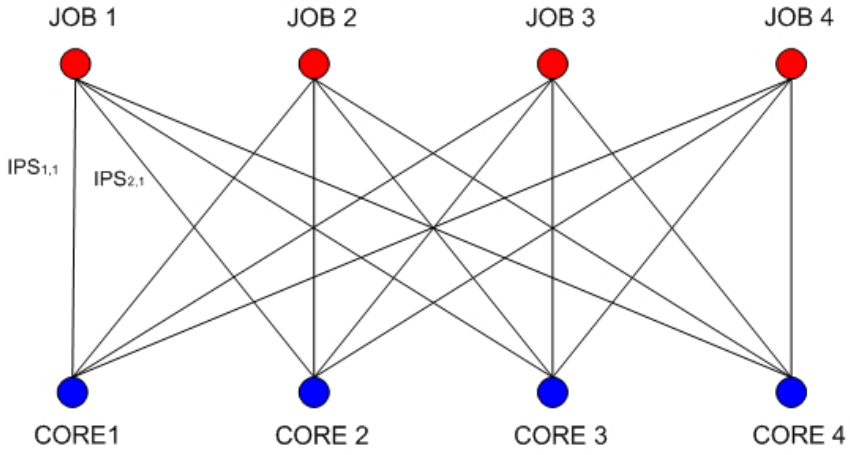


Figure 31: $K_{4,4}$ complete bipartite graph, symbolizing the possibilities of mapping jobs onto cores

If the number of jobs m is smaller than the core number n, we can treat this as if there are m real jobs with nonzero power and $(n - m)$ jobs with zero IPC and zero dynamic power. The algorithm can be used without any major modification. If the number of jobs m is bigger than n, the state-of-art OS scheduler will always select n jobs out of m ones and

assign them onto the CMP, in one scheduling interval. The principle behind this is to keep all the cores busy, but not to make tasks compete for resources on one core. Therefore, we can still think there are currently n jobs on n cores, and solve the problem in a similar way.

4.3.3 Preparation of input to MBM

The key of the success of the MBM algorithm lies in the prediction of IPS. If the future IPC of a job and the future frequency F of the related core do not change in the interval, the IPS is computed as $IPC * F$. However, F may change due to the fine-granularity HW DVFS happening inside the interval, and IPC may change due to the program behavior of the job itself. If IPC and F change, the whole scheduling interval is discretized into n tiny steps, with the job's IPC (IPC_k) and core frequency F_k unchanged in each time step Δt. The average throughput is then $(\sum_{k=1}^{n} IPC_k F_k)/n$. We can see that the prediction of IPS requires the necessary information such as the job's IPC characteristic, the core's voltage and frequency levels, and how much DVFS the core will undergo in the next scheduling interval. To understand the IPC characteristic of a job, modern CMPs usually provide hardware performance counters to measure it. The multiple voltage and frequency levels supported by each core should be provided by hardware manufacturers in the device drivers. *cpufreq* [82] is such a driver program in current versions of Linux to provide this information for underlying CPUs. Finally, how much DVFS a core will undergo is tightly linked with the current temperature and the future thermal change.

The matrices in Figure 32 illustrate the workflow of preparing the input of future IPS for the MBM algorithm. We will first predict the future temperature change without the impact of DVFS when one job is assigned onto one core. Each entry in the Temperature Prediction Matrix on the left of Figure 32 represents the possible future temperature of one core. Based on the knowledge of the current temperatures and the predicted future temperatures, we use a trained table to estimate the average frequency of each core in the next scheduling interval, as shown in the middle of Figure 32. Using the historical IPC information of the job and the predicted core frequency, we can estimate the throuhgput of every possible binding of core

and job. The matrix shown on the right of Figure 32 is the input to the maximum bipartite matching algorithm, which enables the algorithm to run and output the optimal job-core bindings.

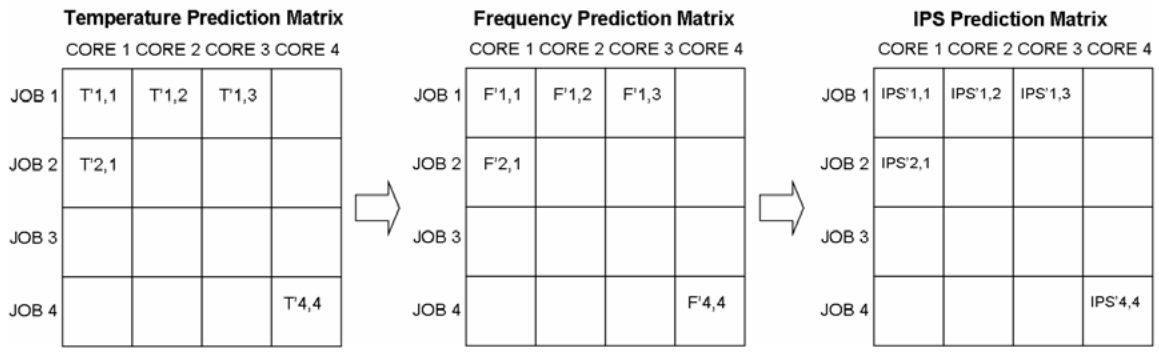


Figure 32: The matrices generated for throughput prediction.

4.3.3.1 Predicting future temperatures The first step to do IPS prediction is to predict the thermal changes in the next interval. The lateral heat conduction between cores in a typical die is very weak compared to the vertical heat conduction to the heat sink [80]. For example, for a typical $32nm$ technology node CMP having each core with $500\mu m$ in thickness and $6mm * 6mm$ in size of area(the area size close to one core in the recent Clarkdale [81], the lateral thermal conductance of the core is only 1/12 of the vertical thermal conductance. Hanumaiah et al. also give a detailed analysis of the relationship between on-die lateral and vertical conductance in [28], and they draw the similar conclusion as ours. Therefore, in order to simplify the computation, the impact of lateral heat conduction is not considered when predicting the temperature. We also conducted experiments on our simulated 16-core CMP by setting the parameters of the cores to be the same and running identical jobs from the same start. We observed that the peak temperatures on the cores were very close at all the time. If the lateral heat conduction was a significant factor, the cores in the middle of the die would be much hotter than the cores on the edge. Even if the core temperature is affected by surrounding cores, as a result of the die and package configurations different from

ours, as the phenomenon observed in [19, 21], there are ways to consider the impact from several surrounding cores, such as the Neural Network predictor shown in [21]. To illustrate the effectiveness of the MBM algorithm, we assume in this work that the future temperature of one core is only strongly correlated with the power inside that core.

By using the lumped model to treat each core as one thermal node, we borrow the classical thermal equation 4.11 from [61, 26] to describe the estimation of future temperature when job j is assigned onto core i.

$$T'_{i,j} = AT_i + BP_i^{leak} + BP_{i,j}^{dyna} \tag{4.11}$$

In this equation, T_i represents the current temperature, and $T'_{i,j}$ is the temperature of the core i after a scheduling interval. A and B are precomputed thermal constants whose actual values depend on the physical behavior of the chip [26, 12], and other parameters such as the heat sink conductance and the scheduling interval length. P_i^{leak} is the average leakage power of core i during the scheduling interval. $P_{i,j}^{dyna}$ corresponds to the dynamic power on core i due to the activities of job j. Here is a simple introduction about how these parameters can be attained in a real system.

- T_i: The current temperature T_i can be read from the hardware thermal sensor on each core, as in [70], or be calculated by using software thermal sensor proposed by W. Wu et al. in [74].
- $A\,and\,B$: The parameters A and B for a real chip can be computed using system identification methods such as the methods presented in [70, 73]. Basically these methods utilize the values of temperatures and power to deduce the values of A and B.
- P_i^{leak}: P_i^{leak} is determined by the temperature and the PV characteristic of core i. The PV characteristic of the core is a parameter that needs to be given by chip manufacturers. E. Kursun et al. in [43] provide a black-box method where they use the thermal imaging technique to estimate the PV characteristics of a CMP. Because the variation of temperature in $8ms$ is relatively small, at most several degrees, the leakage power of one core during such a short scheduling interval can be regarded as a constant. However, when

the interval length increases, the temperature change could be significant. Assuming a constant leakage power is no longer accurate. So in this case, we need to split the whole interval into many smaller intervals, each of which is $8ms$ long. And Equation 4.11 needs to be used over and over. Fortunately, the overhead of computation is distributed into all the smaller intervals, and it equals to the overhead in $8ms$.

- $P_{i,j}^{dyna}$: Inside each program phase of job j, the dynamic power of job j is relatively stable. This is recently shown by the thermal predictability of a majority of SPEC06 benchmarks in [50]. Because our scheduling interval is much smaller than a program phase, we can use the power in the last interval to predict the power in the next interval, without losing much accuracy. We conducted an offline study of predicting future power of SPEC06 benchmarks [91]. The result shown in Figure 33 shows that the average error rate by using the last-value prediction method is around 7%. For a temperature rise of 5°C, the power prediction error can cause a temperature estimation error of 0.18°C in our die and package configuration, suppose the leakage power is as significant as the dynamic power.

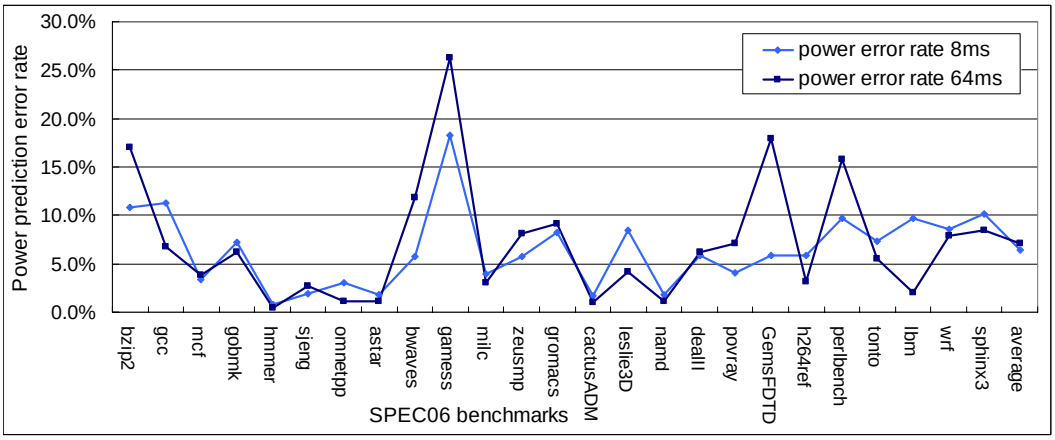


Figure 33: The error rate of power prediction by using last-value prediction method.

By using equation 4.11, we can fill out the diagonal entries in the Temperature Prediction Matrix in Figure 32. The next step is to fill out the remaining entries. The core frequencies

on core k and j are different, so the dynamic power varies when the same job j is assigned onto these two cores. But we can safely assume that the dynamic power of the circuits has a linear relationship with the frequency, suppose the frequency only varies within a small range. This assumption holds correctly especially for the high-IPC jobs, and these jobs have a larger impact on the throughput than the low-IPC jobs that are often memory-bound. Based on this assumption, when job j moves from core k in the last interval to core i in the next interval, the future dynamic power can be estimated as $P_{i,j}^{dyna} = P_{k,j}^{dyna} * F_i/F_k$. Therefore, equation 4.11 changes to:

$$T'_{i,j} = AT_i + BP_i^{leak} + BP_{k,j}^{dyna} * F_i/F_k \tag{4.12}$$

With this equation, we can fill the remaining entries in the Temperature Prediction Matrix.

Our way of temperature prediction is a distributed method. At the beginning of one scheduling interval, each core i first computes the intermediate AT_i, BP_i^{leak} and $BP_{i,j'}^{dyna}$, where j' is the job on core i in the previous interval. The computation involves $3n$ multiplications. Next, the cores send the intermediate results to the core where the OS task scheduler resides. Finally, the task scheduler scales and adds these intermediate results to fill the matrices, taking $n^2 - n$ multiplications and divisions, and $2n^2$ additions.

For more accurate thermal modeling in our experiments, the core is divided into funtional unit blocks. For example, a P4 Northwood core is composed of 24 functional unit blocks. A, B, P, T in the equation 4.11 and 4.12 will become matrices and vectors. We measured that the total time for matrix-vector multiplications on the local core is $25\mu s$ by using a real system with a P4 Northwood core. The scaling and addition of the temperature vectors for all the entries by the scheduler takes about $2.3\mu s$.

4.3.3.2 Frequency prediction The second step for IPS prediction is to predict the future frequencies. Figure 34 depicts how to estimate the frequency. The solid line in the left half of Figure 34 illustrates that the thermal rise suppressed by DVFS in a real scenario. Without DVFS the temperature can rise from initially being below the threshold to some

point above the threshold at the end of the scheduling interval, as indicated by the dotted line. Although the dotted line is an exponential curve, we can approximate it with a linear line with little loss in accuracy, because our scheduling interval is short and on the order of milliseconds to tens of milliseconds. The intersection of the linear line and the thermal threshold θ determines the starting point of the thermal fluctuation. In the right half of Figure 34, we use α to denote the portion before the fluctuation starts; $1-\alpha$ denotes the period when the temperature rises and falls with DVFS off and on. There is a technique described in [34] enabling very fast DVFS switching (on the order of tens of nanoseconds). The technique enables the fine-granularity DVFS, and the duration of DVFS can be very small compared with the task scheduling interval length. If the ratio of time in DVFS during the fluctuation stage is denoted as β, the predicted effective F' in the next interval can be computed using equation 4.13. However, since β is a value linked with the dynamic power, the leakage power and the intensity of DVFS at the specific moment, it can not be obtained easily.

$$F' = \alpha F_{full} + (1-\alpha)(\beta F_{scaled} + (1-\beta)F_{full}) \tag{4.13}$$

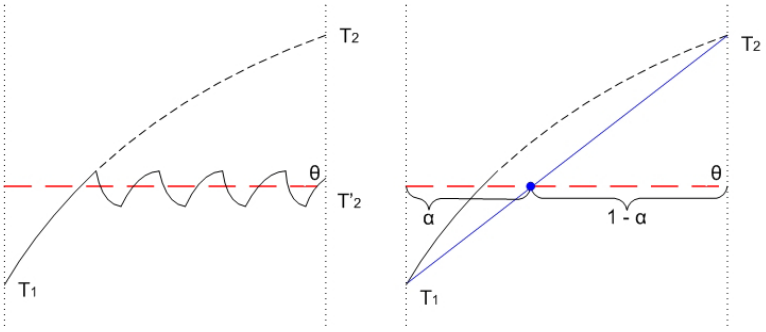


Figure 34: The impact of DVFS on the temperature and the linear interpolation of the temperature

Here we introduce a table-checking method. The method covers all the possibilities when the current temperature and the predicted future temperature are at the different sides of the

thermal threshold. Suppose there is no DVFS in one interval (t_1, t_1'), the current temperature T_1 can change to the future temperature T_1' at the end of the scheduling interval. If there is DVFS, we mark the number of DVFS triggered as k_1. Suppose in another scheduling interval (t_2, t_2'), T_2 at the beginning of the interval changes to T_2' at the end, and the times of DVFS triggered during (t_2, t_2') are k_2. If there are $T_1 = T_2$ and $T_1' = T_2'$, we can assert that $k_1 = k_2$. The reason is that the process of $T_1(T_2)$ changing to $T_1'(T_2')$ reflects the underlying dynamic power and leakage power. If the start and end temperatures are decided, the dynamic and leakage power in this interval can be deduced according to Equation 4.12. The temperature change during this interval is decided and so is the number of DVFS triggered. Therefore, we can conclude that the current temperature and the predicted future temperature determine the actual frequency during this interval. And this is the theory behind our table-checking method.

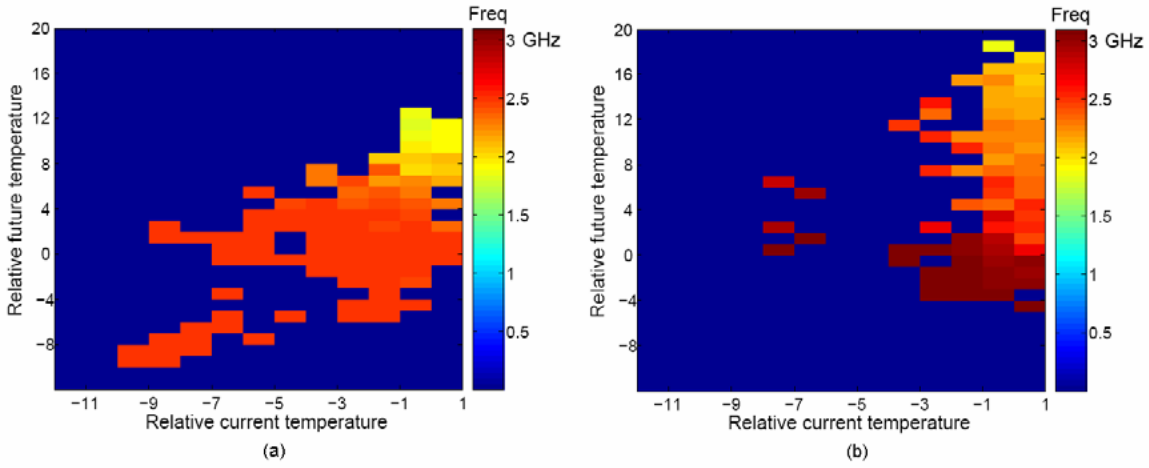


Figure 35: The relationship among current temperature, predicted future temperature, and future frequency((a)die 3, core 7, 2.48GHz;(b)die 3, core 13, 3.06GHz.)

In practice, we logged down the triple data of the current temperature, the predicted future temperature, and the actual future frequency on each core. We then used a 13 by 32 table to remember this relationship among them. The row index of the table marks the small range the current temperature of the core falls into. The column index of the table

marks the small range the predicted future temperature of the core falls into, as the x and y axis in Figure 35 show. The values of temperatures shown in Figure 35 are all relative to the thermal threshold of the core. The value in each table entry corresponds to the actual future frequency on the particular core, shown in Figure 35 as a small colored rectangle. The space overhead of such a table is less than 0.5KB if the values in the table can be compressed. We can then use the trained table on each core to guide the frequency prediction. It takes several seconds to fill the matrices shown in Figure 35(a) and Figure 35(b) with usable values. However, our scheduling algorithm can run for a much longer time to compensate for this training overhead. By using the trained table on each core, we can directly fill the matrix in the middle of Figure 32.

4.3.3.3 An evaluation of temperature and frequency estimation error The accuracy of the predicted frequency depends on the accuracy of the predicted temperature and the accuracy of table-checking method. To understand the combined effect of these factors, we conducted experiments to compare the predicted frequency to the actual frequency in the next scheduling interval. Figure 36 shows the error rate of frequency estimation under different scheduling intervals. When the scheduling interval is $8ms$, the average error rate is 1.7%. For a core with an actual frequency of 3GHz in the next interval, the estimated frequency is around 2.95GHz to 3.05GHz. When the interval is $64ms$, the average error rate reaches 4.26%. The estimated frequency is approximately 2.88GHz to 3.12GHz. The reason for the increase of the prediction error lies in the the power prediction error. The power prediction error in the $64ms$ interval is comparable to the error in the $8ms$ interval, as shown in Figure 33. However, because the thermal coefficient matrix B in Equation 4.11 under the $64ms$ interval is different from the B under the $8ms$ interval, the power error in the $64ms$ interval can generate a larger temperature prediction error. Though such inaccuracy can affect the effectiveness of *MBM*, we show later in the experimental results that *MBM* still wins over other algorithms by quite a large margin.

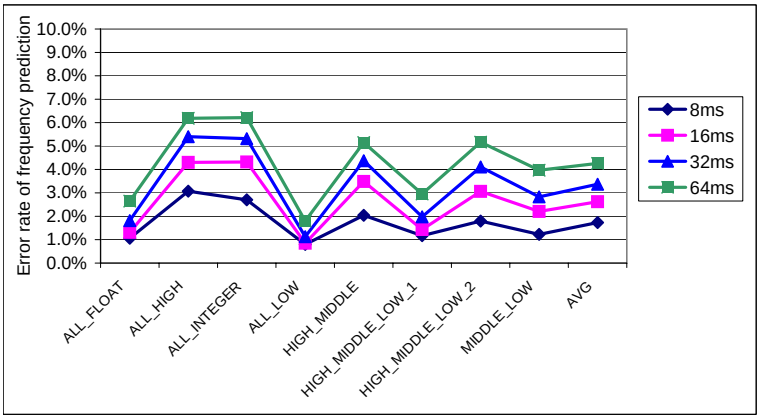


Figure 36: The relative error rate of future frequency prediction under varied scheduling intervals when the number of jobs is 8 and the thermal environment is hot.

4.3.3.4 IPS prediction For the last step, we also use the last-value based IPC prediction method. We conducted the experiment to evaluate this method. The result in Figure 37 shows that the average error rate of IPC predition using $8ms$ scheduling interval is 10.7%. The average error rate when the interval is $64ms$ is 12.3%. Such error rates may generate a large negative impact on throughput in the real situation. Suppose the job-core mapping achieved by using the last-value IPC prediction method is $MAP = \{(i,j)|\ job\ j\ maps\ to\ core\ i\}$, and the other mapping achieved by using oracle knowledge of the future IPC of the jobs is MAP'. The throughput using MAP' is always larger than that using MAP. However, Figure 38 shows that the difference of the throughput between MAP and MAP' is less than 0.08% with the interval length varying. We realize that the IPC misprediction may not change the relative sequence of jobs' IPC. The high-IPC job still has a large chance to be assigned to a fast core, and vice versa. So by the using last-value based method, we computed the predicted throughput(IPS), and can fill the matrix on the right of Figure 32.

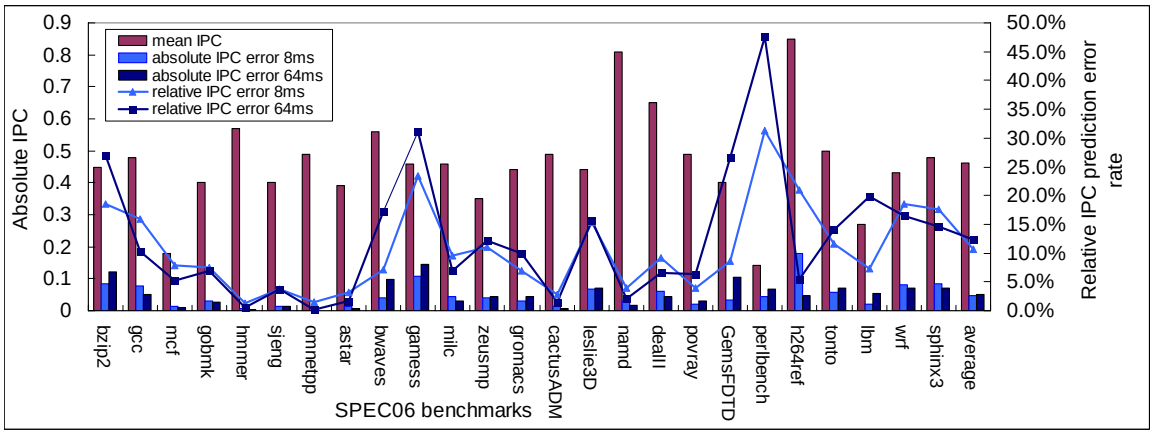


Figure 37: The absolute and relative error of IPC prediction by using last-value prediction method.

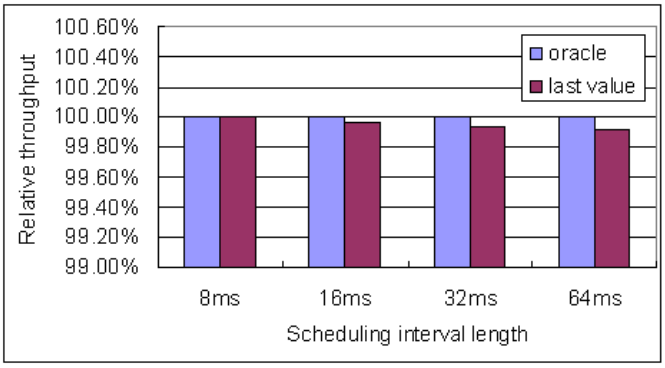


Figure 38: The comparison between the future throughput achieved by using oracle IPC knowledge and the future throughput achieved by using last-value IPC in MBM.

4.3.3.5 Algorithms used in comparisons To demonstrate the effectiveness of our MBM algorithm, the algorithms we compare it with are:

- *Fixed*. This simulates the Linux scheduler and dispatches jobs onto cores randomly in the beginning. The jobs then stay on the cores until the end of execution. It does not consider the benefits that can be brought by higher-frequency cores. Nor does it migrate jobs from the hot cores to the cool cores even if these cores frequently trigger DVFS.
- *NoDVFS*. The *Fixed* algorithm with no thermal constraints and no DVFS is used as a reference in some of our experiments.
- *AppIPC*. By always mapping higher-IPC jobs to faster cores, this achieves the highest throughput when there are no thermal constraints. However, it incurs more DVFS with thermal constraints. And it has task migration overhead.
- *Random*. This algorithm randomly migrates jobs after each scheduling interval. It has the ability to avoid thermal emergencies to some extent, but it doesn't enjoy the benefits brought by fast cores.
- *ThreshHot*. ThreshHot is a single-core algorithm in Section 4.1 that tries to keep hot cores as hot as possible, as long as not too hot to trigger DVFS, and leaves cool cores as a haven for extremely power-intensive jobs. We revised it to adapt to the case of a CMP. It is very effective in avoiding DVFS, but can not take advantage of fast cores.

4.3.4 Experimental setup

4.3.4.1 Floorplan The hardware requirement for our experiments is a CMP aware of process variation. Chip manufacturers currently unfortunately do not disclose process variation parameters for CMPs. So we decided to conduct our experiments by simulation. We modeled 16 cores similar to the P4 Northwood on one die to form the CMP. Because the technology node in P4 Northwood is in 130nm and our simulated CMP uses the 32nm technology node, we first need to shrink the size of each core, the original size of which is $130mm^2$. There are data showing that Intel's recent 32nm-technology Clarkdale processor [81] has an area of $81mm^2$, which accommodates 380 million transistors($4.7\ Million\ transistors/mm^2$).

We decided to use an area of $11.7mm^2$ for each core in order to accommodate 55 million transistors found in the P4 Northwood [89], by using the similar transistor/area ratio.

In the floorplan shown in Figure 39, each P_i represents a core similar to P4 Northwood(without the L2 cache). Each M_i represents the area reserved for shared L2 cache and other shared resources such as memory controllers, routers and power control units. Our floorplan has enough accuracy for thermal simulations, because 22 function units in P4 Northwood, such as execution units and register files, are modeled in our experiments. Current off-the-shelf CMPs can report the temperatures of the individual cores, by putting one thermal sensor on each core. So we still use the temperature of Integer Register Files, one of the hottest function units on the die, as the representative temperature for each core. In this way, we simulate the current placement method of on-die thermal sensors.

Considering that the TDP(Thermal Design Power) of the Clarkdale duo-core processor is 73W, the power traces we collected are scaled to adapt to the much smaller core size.

We made the same assumption as in [68,32] that the shared L2 cache run at a unified frequency, while each core runs at its distinct frequency. We call this variation-aware CMP [68] or CMP-PV. The techniques for implementing the CMP-PV do exist. For example in Intel's Montecito [20], each core has its own clock and V_{dd}, and is called as a voltage and frequency island(VFI). Each VFI contains a clock divider to create its own local clock signal from the output of the shared PLL. As in AMD's quad-core Opteron [67], asynchronous queues provide interfacing between different clock domains, with the buffers between the cores and their routers implemented as dual-clock FIFOs.

4.3.4.2 PV modeling In our experiments, we use VARIUS [59], a statistical tool for modeling variations in micro-architecture. VARIUS models two key process parameters, the transistor threshold voltage V_{th}, and the effective gate length L_{eff}.

We divide our $1.15cm * 1.15cm$ 16-core CMP die into $300 * 300$ grids for the sake of accuracy. To generate the distribution of V_{th} and L_{eff}, we need to set the mean value μ, the standard deviation σ and the spatial correlation range ϕ of V_{th} and L_{eff} in VARIUS.

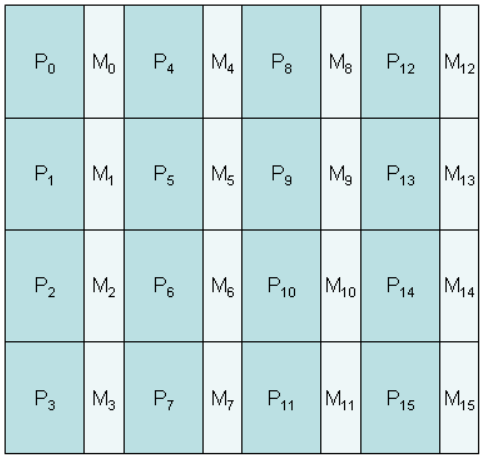


Figure 39: The simulated floorplan of CMP-PV.

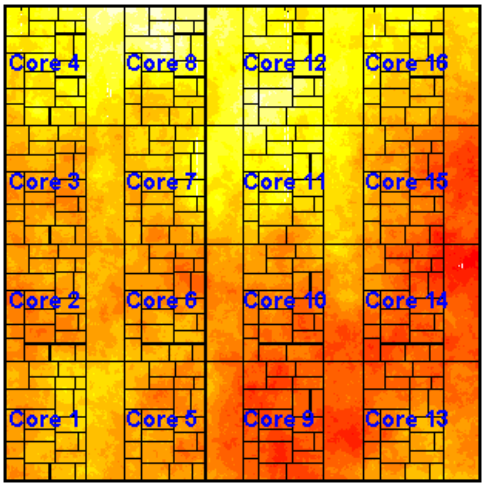


Figure 40: A 16-core CMP with process variation. The colormap under the floorplan shows the within-die variation of the threshold voltage.

The technology size of the transistors in our experiments is $32nm$. Consulting [68, 59], we use $\frac{\sigma}{\mu} = 12\%$ as the intra-die V_{th} variation parameter, $\frac{\sigma}{\mu} = 6\%$ as the intra-die L_{eff} variation parameter, and $\phi = 0.5$ for spatial correlation range, taking the size of our CMP into consideration. Since V_{th} and L_{eff} have a very strong correlation [59], we can use V_{th} to replace L_{eff} in all of the calculations that evaluate the variation of frequency and leakage power. Figure 40 shows the resulting 16-core CMP die. Each grid corresponds to one colored dot in the background and has a distinct V_{th} and L_{eff} value.

After modeling the distribution of V_{th}(and L_{eff}), its impact on leakage power can be examined. The variation of V_{th} directly affects the leakage current, I_{leak}, of the transistors. We assume the temperature is uniform inside one functional unit(FU) in one core, and then I_{leak} at each grid can be aggregated to get the the leakage power of the function unit. The sum of the leakage of all the units then corresponds to the leakage power of one core. It is reasonable to assume such thermal uniformity for two reasons. The typical FU of logic is small, so there is little thermal variation in it. For a large FU such as a cache memory, the power density and the corresponding temperature is not high, so the temperature is not the major factor affecting the leakage power variation in cache. Figure 41(a) shows the leakage power discrepancy caused by process variation in 600 sample dies at $T = 100°C$. The cores in the 16-core CMP on average consume 40% more leakage power than the least leaky core in the same die.

Another impact of V_{th} and L_{eff} variation is on the maximum frequency of each individual core. V_{th} and L_{eff} determine gate delay, which affects the critical path delay in the core. Assuming every grid might contain a critical path, and each critical path consists of n_{cp} gates, the critical path delay T_{cp} can be calculated. The longest critical path delay in one whole core is then $max(T_{cp})$, and the core frequency is estimated to be the inverse of the longest path delay $1/max(T_{cp})$ [10]. Figure 41 (b) shows the frequency discrepancy caused by process variation in 600 sample dies. The cores on average are 10% faster than the slowest core in the die.

The modeling results that we get are consistent to the modeling results shown in [32,68]. We select 20 sample dies to run our experiments. In our simulation, the cores in our 16-core CMP can run at distinct maximum frequencies, and the other parts on the die run at a unified frequency.

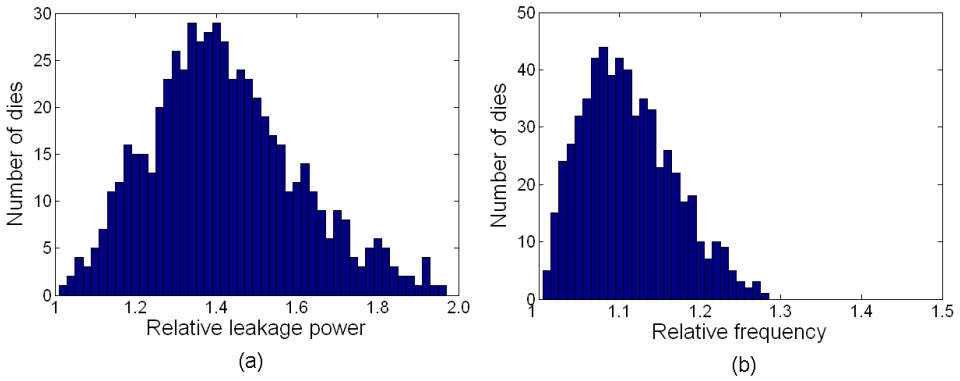


Figure 41: Histograms of the ratio between (a) the average leakage power of the cores and the power of the least leaky core (b) and between the average frequency of the cores and the frequency of the slowest core in the die.

4.3.4.3 Simulation tools and benchmarks To enable the power trace driven simulation, we use HotSpot [61, 62] as the thermal simulation tool. The 4th-order Runge-Kutta method in HotSpot is replaced by TILTS [26] to generate accurate temperatures at faster simulation speeds. The advantage we have is that we can collect power and IPC traces from a real system using the Linux 2.6 kernel and a P4 CPU. This methodology of power collection is justified in [73]. Next, the benchmarks from SPEC06 are classified as HIGH, MEDIUM, LOW according to their IPC characteristics, as shown in Table 4. The variation among the IPC of the jobs can let us have a deep understanding of the effectiveness of the algorithms, because some of these algorithms utilize IPC information to maximize the throughput. However, even if the jobs are the same on the CMP, they may run at different program phases. So task scheduling will still provide opportunities for performance improvement. Each trace

is reduced to several seconds long from a complete run of the benchmark, which is still long enough to exhibit representative power and IPC characteristics. Traces are selected and combined into multi-programmed workloads to be used as input to HotSpot. Table 5 shows how we select representative benchmark combinations. The main idea is to mingle the jobs with different IPC characteristics.

We modified HotSpot to implement the task execution, migration and OS scheduling schemes. For each workload, Hotspot runs for a duration equal to 5 seconds of wall time, and the results such as throughput are collected. Because the results could be different for the dies with different PV variation, we selected 20 dies randomly and show the average results in Section 4.3.5.

We set the thermal threshold as 76C, 71C, 66C and 61C respectively, to simulate the thermal environments marked as mild, hot, severe, and extremely hot. A too high threshold does not make the cores trigger DVFS. A too low threshold makes the cores trigger DVFS all the time. Apparently, the algorithms in comparison will make little difference under such too high or too low thresholds.

Table 4: IPC characteristics of benchmarks in SPEC06

h264ref 0.85	namd 0.81	dealII 0.65	hmmer 0.57	bwaves 0.56
tonto 0.50	omnetpp 0.49	povray 0.49	cactusADM 0.49	sphinx3 0.48
gcc 0.48	gamess 0.46	milc 0.46	bzip2 0.45	gromacs 0.44
leslie3d 0.44	wrf 0.43	gobmk 0.40	sjeng 0.40	GemsFDTD 0.40
astar 0.39	zeusmp 0.35	lbm 0.27	mcf 0.18	perlbench 0.14

4.3.4.4 Overhead Several factors of overhead need to be considered inside or between scheduling intervals. The smallest scheduling interval in our experiments is $8ms$. It is chosen since it is close to the thermal constant of the chip, and also close to the lower bound of

Table 5: The combination of benchmarks when the number of jobs is 8.

HIGH_MIDDLE_LOW_1	h264ref namd omnetpp povray gamess leslie3d astar zeusmp
HIGH_MIDDLE_LOW_2	bwaves omnetpp milc games leslie3d gobmk zeusmp mcf
HIGH_MIDDLE	h264ref namd bwaves tonto cactusADM sphinx3 games bzip2
MIDDLE_LOW	milc games leslie3d wrf GemsFDTD astar mcf perlbench
ALL_HIGH	h264ref namd dealII hmmer h264ref namd dealII hmmer
ALL_LOW	zeusmp lbm mcf perlbench zeusmp lbm mcf perlbench
ALL_FLOAT	bwaves games milc zeusmp gromacs cactusADM leslie3d namd
ALL_INTEGER	bzip2 gcc mcf gobmk hmmer sjeng omnetpp astar

Linux recommended scheduling interval length. Furthermore, we can get accurate power traces from the real P4 chip in such a small interval.

HotSpot is modified to simulate the hardware DVFS mechanism. Our simulated DVFS mechanism is reactive, and the usage is to curb thermal trespassing. When the on-core thermal sensor senses the trespassing of core temperature, it triggers DVFS, and the DVFS lasts for 800 μs before switching off. During the triggering, the frequency and voltage of cores are both lowered to the 60% of the maximum scale. We tried other DVFS levels(70% or 80%) and found they do not affect the relative effectiveness of all the algorithms in comparison. Each DVFS switching charges an almost negligible penalty of $100ns$, simulating the efficient on-chip regulator implementation [40]. We admit there are other dymamic thermal management techniques, such as Decode Throttling and I-cache Toggling in [11]. But they will also affect the core performance when triggered. The only difference is that they are at the micro-architecture level.

After one scheduling interval, each core conducts their local matrix-vector computation, as described in Section 4.3.2. The centralized scheduler then collects all the necessary information and runs the scheduling algorithm. For distributed temperature computation, the time penalty of $28\mu s$ is charged for every $8ms$. The scheduler running the *MBM* algorithm needs $16\mu s$ to get an optimal scheduling solution.

If the job needs migration, the migration penalty for each task migration is set to $100\mu s$, which refers to [17]. We assume the L2 cache is shared, so $100\mu s$ is long enough for a job to conduct a context switch and move the content in L1 cache. For example, moving 96kB data in IBM's POWER5 L1 cache takes only $42\mu s$ in a modern 3GHz network on chip when the conjestion rate is not high. And we measured the average context switch time in P4 Northwood processor as $35\mu s$.

During the period when any of the aforementioned overhead happens, no useful instructions in the workloads can be executed. Therefore for each job on the core, if the migration happens after an $8ms$ interval, the total overhead will be $144\mu s/8ms = 1.8\%$; if no migration happens, the overhead will only be $44\mu s/8ms = 0.55\%$.

4.3.5 Results

We conducted experiments to compare the performance of all the aforementioned algorithms, by varying parameters such as the number of jobs on CMP, thermal threshold, and interval length. In this section, we introduce the results. We will also give measurement on the algorithms' overhead and their energy consumption.

4.3.5.1 DVFS and throughput Figure 42 and Figure 43 show the relative throughput achieved and DVFS triggered by running varied number of jobs when the interval is 8ms and the thermal environment is hot. Compared with the reference throughput without thermal constraints(*NoDVFS+Fixed* in Figure 30), *Fixed* reaches 88.6%, 87.0%,85.9% and 82.3% of the reference throughput. *AppIPC* is worse than *Fixed* when the CMP is fully loaded, because it triggers more DVFS. However, it shows higher throughput than *Fixed* when the number of jobs decreases. The reason is that the jobs in *AppIPC* concentrate on faster cores, so they run faster when DVFS is not triggered.

Surprisingly, *Random* achieves higher throughput than *Fixed* and *AppIPC*. This can be explained by the smaller number of DVFS it triggers, shown in Figure 43. *Random* achieves 13% to 95% reduction of DVFS in *AppIPC*. However, because high-IPC jobs may

not be assigned to the faster cores in *Random*, the higher frequencies in faster cores may not be utilized. That is why *Random* only results in 0.4% to 3.8% throughput improvement compared with *AppIPC*.

In Figure 43, *ThreshHot* reduces DVFS triggerings by 11.6%, 14.1%, 9.1% and 2.1% respectively from *Random* when n decreases. Moreover, *ThreshHot* always tries to keep hot cores hot, and leave cool cores cool. Because hotter cores are typically faster cores, it implicitly utilizes a faster core more frequently than slower cores. So its throughput is 1% higher than *Random* when $n = 16$, and 4.1% higher when $n = 4$. The weakness of *ThreshHot* is that it may not bind high-IPC jobs to faster cores, even when it utilizes faster cores often.

Our *MBM* algorithm tries to maximize the overall throughput in the CMP. It tends to avoid the unwanted mappings that trigger DVFS, or assign high-IPC jobs to slow cores. Though it triggers the smallest amount of DVFS compared with other algorithms, the amount of DVFS is very close to those in *ThreshHot* in Figure 43. So we believe that mapping jobs to cores when the benefit outweighs the loss, makes it get higher throughput than *ThresHot*. When the number of jobs $n = 16, 12, 8, and\ 4$ respectively, it gets 0.1%, 0.7%, 1.2% and 1.6% higher throughput than *ThreshHot*.

Another interesting phenomenon we observe from Figure 42 and Figure 43 is that the smaller the number of jobs is, the better *MBM*(and *Random* and *ThreshHot*) works. When the number of jobs n becomes smaller, selecting n cores from a total of m cores can generate larger variation in the average core frequency. So the selection decision has a larger impact on the throughput. This phenomenon is also observed in [68].

4.3.5.2 Detailed throughput for different workloads

Figure 44 and Figure 45 display the relative throughput achieved and absolute number of DTM triggered by different workloads when the number of jobs is 8, the interval is 8ms, and the thermal environment is hot. For almost all the workloads, the sequence of algorthm effectiveness is *MBM* > *ThreshHot* > *Random* > *AppIPC* > *Fixed*. There are some details that cannot be ignored. For example, *Random* sometimes generates lower throughput than *AppIPC*, e.g., in ALL_LOW

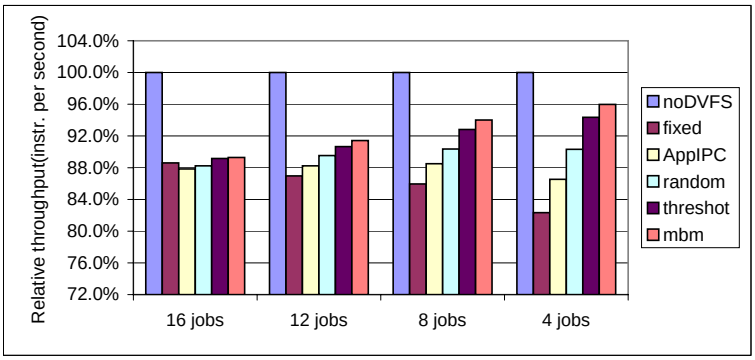


Figure 42: The relative throughput achieved by running varied number of jobs when the interval is 8ms and the thermal environment is hot.

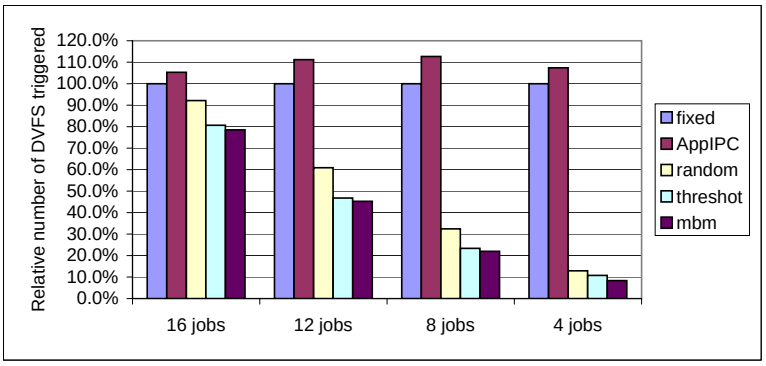


Figure 43: The relative DVFS triggered by running varied number of jobs when the interval is 8ms and the thermal environment is hot.

and HIGH_MIDDLE_LOW_2. This shows the unstableness of *Random*. *MBM* achieves the higher throughput by 10.3% and 7.6% respectively than *Fixed* and *AppIPC* in the workload ALL-HIGH. *MBM* also achieves 11.2% and 8.0% higher throughput than *Fixed* and *AppIPC* in HIGH_MIDDLE_LOW_1. When the jobs are all low-IPC ones in ALL-LOW, *MBM* is only 3.8% and 1.2% better than *Fixed* and *AppIPC*. This can be explained by the small amount of DVFS triggered in *Fixed* and *AppIPC* in Figure 45.

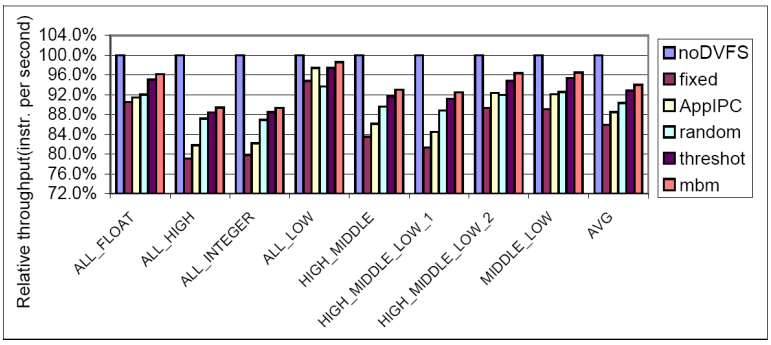


Figure 44: The relative throughput achieved by different workloads when the number of jobs is 8, the interval is 8ms, and the thermal environment is hot.

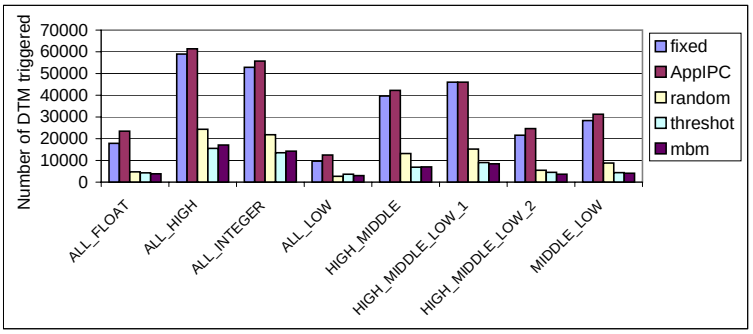


Figure 45: The number of DVFS triggered by different workloads when the number of jobs is 8, the interval is 8ms, and the thermal environment is hot.

4.3.5.3 Thermal environment *MBM* can be effective under a wide range of thermal environments. Figure 46 shows the throughput of the CMP under different thermal thresholds when the number of jobs is 8. The severeness of the environment can be seen from the extent of the throughput decrease. In Figure 46, the throughput in *AppIPC* corresponds to 96.5%, 88.5%, 73.9% and 61.8% of *NoDVFS*(*Fixed* with no thermal constraints) under different thermal conditions. *ThreshHot* beats all the other algorithms in all the environments. More specifically, it achieves 5.5% and 5.4% higher throughput than *AppIPC*, when the thermal environment is hot and severe respectively. When the environment is mild, there is not much DVFS triggered even in *AppIPC*. In contrast to this, when the environment is extremely severe, it is difficult to find a cool core which doesn't trigger DVFS. So in both cases, the effectiveness of *MBM* relative to *AppIPC* decreases. It improves only by 1.9% and 1.3% respectively under these two environments. However, *Random* becomes worse than *AppIPC* under these environments, because *Random* now triggers the similar amount of DVFS as *AppIPC* does.

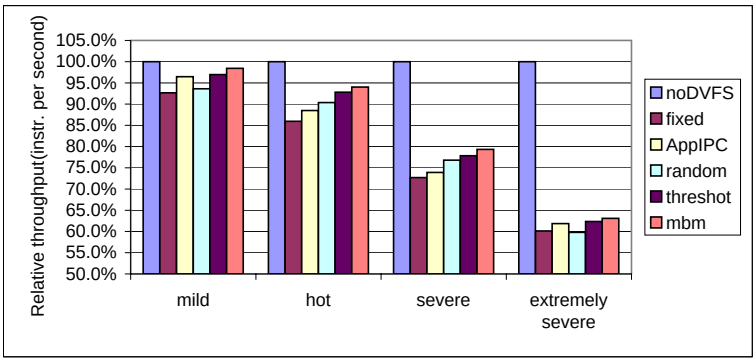


Figure 46: The relative throughput under different thermal environments when the interval is 8ms and the number of jobs is 8.

4.3.5.4 Varied interval length Modern operating systems always use a scheduling inteval length that has virtually no side effect on program performance. And the scheduling inteval length could be varied in different systems. Figure 47 shows the relative throughput

achieved by varying scheduling interval length when the number of jobs is 8 and the thermal environment is hot. We can see that *Fixed* and *AppIPC* are insensitive to the change of interval length. The insensitivity of *AppIPC* reflects that the IPC of a majority of SPEC06 benchmarks are stable during at least $64ms$. Otherwise the results using $8ms$ scheduling intervals will be quite different. *ThreshHot* suffers from the biggest amount of throughput degradation, 3.8% when the interval increases from $8ms$ to $64ms$. The reason is that a high-IPC job can easily raise the temperature close to the threshold in $64ms$. To avoid any thermal trespasses, *ThreshHot* chooses to put this high-IPC job onto a slow but cool core. So it still suffers from big throughput loss due to the low frequency of the core. *MBM* suffers a mild performance loss of 1.9% when the scheduling interval changes from $8ms$ to $64ms$. We believe the degradation in a large scheduling inteval is due to the increased error rate of future frequency prediction, which is discussed in detail in Section 4.3.3.3. The fact that the scheduler can not respond to the phase changes inside a large interval can also be the reason for such performance loss. However, such mild loss can not deny the effectiveness *MBM* over a wide range of interval lengths.

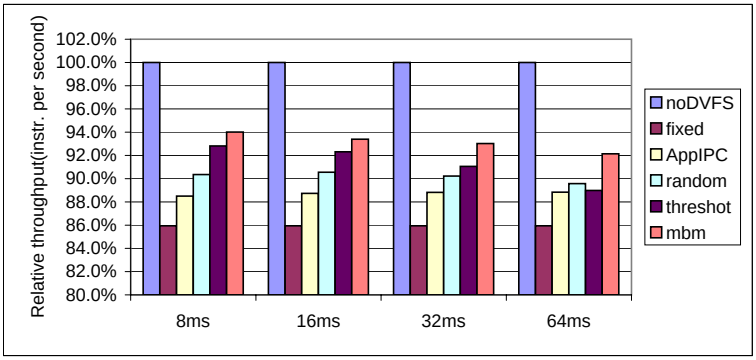


Figure 47: The relative throughput achieved by varying scheduling interval length when the number of jobs is 8 and the thermal environment is hot.

4.3.5.5 Overhead Since our algorithm involves throughput prediction, task migration and scheduling algorithms in relatively small scheduling intervals, we want to evaluate the

overhead incurred. The result is shown in Figure 48. In *Fixed*, there is no task migration and algorithm computation. We found the overhead from DVFS is negligible. The fact that *ThreshHot* and *MBM* incur nearly the same amount of overhead when the interval length is $8ms$, can only indicate that the major overhead is not from our MBM scheduling algorithm. The task migration overhead is an important factor, because *Random* incurs the biggest overhead when the interval length is $8ms$. Besides task migration, *Random* does not have other overhead. *Random*, *ThreshHot* and *MBM* show much bigger overhead than *AppIPC*. It is due to the fact that *AppIPC* does not switch jobs when the IPC of the jobs are stable in a certain phase. The overhead increases when the task migration happens more frequently in the smaller scheduling intervals. Finally, another important source of overhead is temperature prediction in *ThreshHot* and *MBM*. Because the leakage power can only be assumed to be a constant in a very short period of time, the prediction of leakage power requires many rounds of temperature computation to retain the accuracy when the interval length is long. Therefore, when the interval length is $64ms$, a majority of the overhead in *ThreshHot* and *MBM* is due to temperature prediction. The highest overhead happens when the interval is $8ms$. The overall overhead is 1.34% for *MBM* and 1.57% for *Random*.

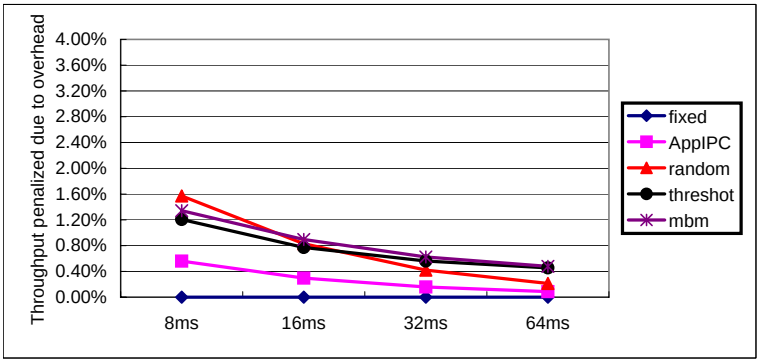


Figure 48: The relative throughput penalized due to all sorts of overhead under different scheduling intervals when the number of jobs is 8 and the thermal environment is hot.

4.3.5.6 Energy consumption per instruction One metric to measure the goodness of modern task schedulers is the power and energy consumption. From the results shown in Figure 49, we found *AppIPC* consumes the smallest energy per instruction(EPI) among all the algorithms. Although *MBM* generates the highest throughput, it results in higher EPI. The EPI in *MBM* is 2.7%, 6.9%, 7.3% and 0.8% higher than *AppIPC* respectively, when $n = 16, 12, 8, 4$. The explanation is that the CMP can benefit from the cubic power reduction when DVFS is triggered. The more DVFS triggered, the smaller power the CMP consumes. In the case of *MBM*, the throughput improvement over *AppIPC* can not compensate for the energy consumed.

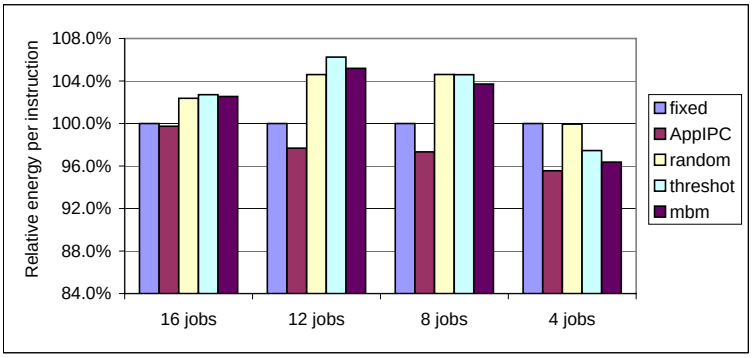


Figure 49: The relative energy per instruction(EPI) by running varied number of jobs when the interval is 8ms and the thermal environment is hot.

5.0 CONCLUDING REMARKS

As IC technology size further scales down, processors endure higher on-chip power density, and temperatures on chip can easily surpass the thermal threshold. High temperatures on the chip, if not controlled, can damage the chip or even burn the chip out. 3D die stacking, as a promising new IC technology to improve CMP performance, can exacerbate the thermal condition, because the power density per unit volumn is dramatically increased. Meanwhile, there is significant within-die process variation in the current and near-future CMPs. Spatial variation of core frequency and leakage power can cause imbalanced thermal distribution on chip, which forms another thermal problem.

Due to the high cost and inefficiency of the current mechanical cooling techniques, one method to constrain a CPU from overheating is hardware dynamic thermal management(HW DTM). One widely used technique belonging to this category is dynamic voltage/frequency scaling(DVFS). However, HW DTMs constrain the temperature at a cost of large performance loss, in terms of longer program execution time, or lower CPU throughput. This thesis addresses such a problem, which is how to eliminate unnecessary hardware-level DTMs and improve chip performance, with the constraint that the processor needs to run under the thermal threshold.

We attacked this problem by proposing software-level task scheduling algorithms, in three different hardware scenarios: a single-core processor; 3D die-stacking processor; and CMPs with significant process variations. In each scenario, the proposed algorithm achieved two goals at the same time: improving the performance by avoiding HW DTMs and meeting thermal constraints.

The contribution of this thesis is that we improved scheduling algorithms to save performance loss, for a large set of current and near-future processors, which suffer from the invisible thermal wall. Specifically, the scheduling algorithm proposed for single-core processors was implemented in a real system and proven to work. Although we used on-demand clock modulation in the single-core processor, and used DVFS in the CMPs as the HW DTM methods, our algorithms will work for any other forms of DTM. In the following, we will summarize the results in our experiments, and discuss about future work.

5.1 SUMMARY OF RESULTS

To improve the performance of single-core processors, we proposed a heuristic algorithm named ThreshHot, which judiciously schedules hot jobs before cool jobs, to make the future temperature lower. Furthermore, it always keeps the temperature as close to the threshold as possible, when the temperature is below the threshold. We conducted experiments on a real P4 CPU with Linux operating system. In all the workloads(combination of the representative jobs in SPEC and non-SPEC benchmarks) and in all the thermal environment, the ThreshHot scheduler consistently removed more DTMs than other existing schedulers, often by a great amount. The DTM 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. By removing unnecessary HW DTMs, the highest performance improvements from the ThreshHot scheduler are seen in the workload "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 reductions of execution time are 3.8%, 4.7%, 4.1%, and 3.25% for mild, medium, harsh thermal environment, and non-SPEC programs respectively, compared to baseline scheduling algorithm in Linux. Considering the slowing-downs of program execution caused by HW DTMs in mild, medium, harsh thermal

environment are only 15.4%, 28.9%, 21.8%, and 16.6% for mild, medium, harsh thermal environment, and non-SPEC programs respectively, the performance improvement achieved by the ThreshHot scheduler is significant.

In 3D die-stacking processors, three heuristics were proposed and combined as one algorithm. First, vertically stacked cores are treated as a core stack. The power of jobs is balanced among the core stacks instead of individual cores. Second, hot jobs are moved close to the heat sink to expedite heat dissipation. Third, when thermal emergencies happen, the most power-intensive job in a core stack is penalized in order to lower the temperature quickly. We called this scheduling algorithm as Balancing-by-stack algorithm, and compared it with Linux Baseline, Random, Round-Robin, and Balancing-by-core algorithm(which was also proposed in the thesis for comparison purpose). The experiment results showed that the Random, RR and Balancing-by-core can reduce the thermal emergency time penalized by HW DTMs by 30.9%, 37.41% and 36.4% on average respectively. Our Balancing-by-stack algorithm removes the most emergency time in 8 cases of 9 benchmark workloads. An average of 46.23% emergency time reduction was observed, with a variation from 6.06% to 96.04%, with respect to the workloads tested. Less thermal emergency time corresponds to fewer HW DTMs triggered, and shorter total execution time of the jobs. On average, the Balancing-by-stack algorithm achieves a 5.11% reduction of the total execution time compared to the Linux baseline, while the Random, RR, and Balancing-by-core algorithm reduce the execution time by merely 1.45%, 1.72% and 1.65% respectively.

In the CMPs affected by significant process variation issues, maximizing the overall throughput on all the cores is in conflict with satisfying on-chip thermal constraints imposed on each core. A maximum bipartite matching(MBM) algorithm was proposed to solve this dilemma, to exploit the maximum performance from the chip. We compared MBM algorithm with Linux baseline algorithm, AppIPC [68], Random, and ThreshHot. When the thermal environment is hot and the number of jobs n = 16, 12, 8, and 4, the MBM algoritm removes 21%, 54%, 77%, and 91% of HW DTMs(DVFS) from the Linux Baseline respectively, outperforming the other algorithms in comparison. In terms of CPU throughput, compared

with Linux Baseline, the MBM algorithm shows 1%, 4%, 8%, and 14% of throughput improvement respectively; compared with ThreshHot, it can still show 0.1%, 0.7%, 1.2%, and 1.6% of throughput improvement respectively. In the experiments, we also simulated various thermal environment and try varied scheduling intervals. The experiment results proved that the MBM algorithm unanimously won over the other algorithms in comparison in terms of CPU throughput.

5.2 FUTURE WORK

This thesis enables a lot of future research in task scheduling and thermal management, and we will look into a few directions that are related to or enabled by this thesis.

5.2.1 Implementation of our algorithms onto real chips

We proposed new software task scheduling algorithms in the CMPs with stacked dies and the CMPs with significant process variation. The results presented in this thesis were based on simulation. It will be very interesting to implement these algorithms in the real chips with stacked dies, or the ones with different maximum frequency and leakage power on each core. Because in real chips there are usually constraints other than the thermal constraints, there may be more challenges or better opportunity for performance improvement.

5.2.2 Self adaptive scheduling algorithms

In this thesis, we have done a limited study of many factors that can affect the performance of the algorithms. The thermal constant of the die is such a factor, and there are other factors such as the migration overhead of the jobs, and the HW DTM overhead. Although we showed that the algorithms we proposed are robust enough to outperform other algorithms when these factors vary, we still need to find the sweet point of the algorithm settings.

It would be very nice to let the algorithm settings, such as the scheduling interval, adapt to the environment, using the environment factors as input. For example, when the thermal enviroment is too hot or too cool, task scheduling will only incur more context switch overhead. The algorithms can choose not to be activated when they receive such a signal about thermal environment in the input. Furthermore, the OS task scheduler can collect task migration overhead information. The information of HW DTM overhead can be written into ROM and be collected by OS. All this information will be sent to the proposed algorithms, which can tune the scheduling inteval on-the-fly, in order to achieve the highest performance improvement.

5.2.3 Online computation of thermal coefficients

The key of computing temperature lies in obtaining accurate thermal coefficients **A** and **B**, as introduced in Section 2.1.3. Currently, when the algorithms are running on the real chips, the values of the coefficients need to be measured and calibrated using the thermal sensor readings from Pentium 4 processor. And we assumed the thermal coefficients are the same for the same type of processors in this thesis. However, the thermal coefficients can be varied even for the same type of processors, because the heat spreader, the heat sink and the cooling fan are different in each machine box. One possible research direction could be online computation of thermal coefficients. For each individual CPU, we can use the CPU activities as the proxy of power consumption. The temperatures can be still read from the thermal sensors on chip. We may deduce the thermal coefficients for any CPU using Equation 2.4 in Section 2.1.3, which is more accurate than applying the same thermal coefficients for a family of processors.

5.2.4 More accurate prediction of power consumption of jobs

We used Last-power-value predictor in this thesis to predict the power consumption of the jobs in the next scheduling interval. Our study showed that the accuracy of the predictor for

a majority of the benchmarks is good. However, the predictor incurs big prediction error for a small number of specific benchmarks. Program profiling, compiler techniques or machine learning algorithms may give us hints to design a better power preditor, which can ultimately enhance the effectiveness of the scheduling algithms proposed.

5.2.5 New optimization objectives

All the algorithms proposed in this thesis aimed at improving CPU performance, including the goals such as shortening the total execution time, or increasing CPU throughput. In the machines such as desktops or workstations, our algorithms can help to improve the productivity of the daily work of people. However, these days smart phones make the daily life of the users much more convenient, and the energy consumption characteristics of the smart phones are always a big concern. There is an urgent need of tailoring our task scheduling algorithms on smart phones, with reducing energy consumption as the primary optimization goal. The existence of thermal constraints on the smart phones, if there is any, will just make the problem even more interesting and challenging.

BIBLIOGRAPHY

[1] M. Awasthi, R. Balasubramonian, "Exploring the design space for 3D clustered architectures", *3rd IBM Watson Conference on Interaction between Architecture, Circuits, and Compilers (P=ac2)*, Yorktown Heights, October 2006.

[2] K. Banerjee, S. Souri, P. Kapur, and K. Saraswat, "3-D ICs: a novel chip design for improving deep-submicrometer interconnect performance and systems-on-chip integration," *Proceedings of the IEEE*, vol. 89, pp. 602–633, May 2001.

[3] N. Bansal, T. Kimbrel, K. Pruhs, "Dynamic speed scaling to manage energy and temperature," *the 45th Annual IEEE Symposium on Foundations of Computer Science,* pp. 520-529, 2004.

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

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

[6] F. Bellosa, A. Weissel, M. Waitz, S. Kellner, "Event-driven energy accounting for dynamic thermal management," *Workshop on Compilers and Operating Systems for Low Power,* 2003.

[7] Bryan Black, et al., "Die stacking (3D) microarchitecture," MICRO 2006: 469-479.

[8] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, "Parameter variations and impact on circuits and microarchitecture," in Design Automation Conference, June 2003.

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

[10] K. Bowman, S. Duvall, and J. Meindl, "Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration," IEEE J. Solid State Circuits, vol. 37, no. 2, pp. 183-190, Feb. 2002.

[11] D. Brooks, M. Martonosi, "Dynamic thermal management for high-performance microprocessors," *the 7th International Symposium on High-Performance Computer Architecture,* pp. 171-180, 2001.

[12] D. Brooks, R. P. Dick, R. Joseph, and L. Shang, "Power, thermal, and reliability modeling in nanometer-scale microprocessors," *IEEE Micro*, 27(3), 2007.

[13] 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 International Symposium on High-Performance Computer Architecture,* 2007.

[14] J. Choi, "Thermal-aware task scheduling at the system software level," ISLPED'07, August 27-29, 2007, Portland, Page 213-218.

[15] Ayse Kivilcim Coskun, Richard Strong, Dean M. Tullsen, Tajana Simunic Rosing, "Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors," SIGMETRICS/Performance 2009: 169-180.

[16] Y. Deng, W. P. Maly, "2.5-dimensional VLSI system integration," *IEEE Trans. VLSI Syst.*, 13(6):668-677, 2005.

[17] James Donald, Margaret Martonosi, "Techniques for multicore thermal management: classification and new exploration," ISCA 2006: 78-88.

[18] J. Dorsey, S. Searles, M. Ciraula, E. Fang, S. Johnson, N. Bujanos, R. Kumar, D.Wu, M. Braganza, and S. Meyers, "An integrated quad-core Opteron processor," *ISSCC 07:IEEE International Solid-State Circuits Conference Digest of Technical Papers*, 2007.

[19] Thomas Ebi, Mohammad Abdullah Al Faruque, Jorg Henkel, "TAPE: Thermal-aware agent-based power econom multi/many-core architectures," *ICCAD* 2009: 302-309.

[20] T. Fischer, J. Desai, B. Doyle, S. Naffziger, and B. Patella, "A 90-nm variable frequency clock system for a powermanaged Itanium architecture processor," *IEEE Journal of Solid-State Circuits*, 41(1), Jan 2006.

[21] Yang Ge, Parth Malani, Qinru Qiu, "Distributed task migration for thermal management in many-core systems," *DAC* 2010: 579-584.

[22] M. Gomaa, M. D. Powell, and T. N. Vijaykumar, "Heat-and-Run: Leveraging SMT and CMP to manage power density through the operating system," *ASPLOS*, 2004.

[23] B. Goplen, S. S. Sapatnekar "Thermal via placement in 3D ICs," *ISPD 2005*, pp.167-174.

[24] B. Goplen, S. S. Sapatnekar, "Placement of thermal vias in 3-D ICs using various thermal objectives," *IEEE Trans. on CAD of Integrated Circuits and Systems 25(4)*, pp. 692-709, 2006.

[25] 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.

[26] Yongkui Han, Israel Koren, C. Mani Krishna "TILTS: a fast architectural-level transient thermal simulation method," J. Low Power Electronics 3(1): 13-21, 2007.

[27] H. Hanson, S. Keckler, S. Ghiasi, K. Rajamani, F. Rawson, J. Rubio, "Thermal response to DVFS: Analysis with an Intel Pentium M," *International Symposium on Low Power Electronics and Design,* pp. 219-224, 2007.

[28] Vinay Hanumaiah, Sarma B. K. Vrudhula, Karam S. Chatha, "Maximizing performance of thermally constrained multi-core processors by dynamic voltage and frequency control," *ICCAD* 2009: 310-313.

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

[30] T. Heath, A. P. Centeno, P. George, L. Ramos, Y. Jaluria, R. Bianchini, "Mercury and freon: temperature emulation and management for server systems," *International Conference on Architectural Support for Programming Language and Operating Systems,* pp. 106-116, 2006.

[31] S. Heo, K. Barr, K. Asanovic, "Reducing power density through activity migration," *International Symposium on Low Power Electronics and Design,* pp. 217-222, 2003.

[32] Sebastian Herbert, Diana Marculescu, "Variation-aware dynamic voltage/frequency scaling", HPCA 2009: 301-312.

[33] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, S. Velusamy, "Compact thermal modeling for temperature-aware design," *the 41st Annual Conference on Design Automation,* pp. 878-883, 2004.

[34] W. Huang, E. Humenay, K. Skadron, M. R. Stan, "The need for a full-chip and package thermal model for thermally optimized IC designs," *the 2005 International Symposium on Low Power Electronics and Design,* pp. 245-250, 2005.

[35] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, and S. Ghosh, "HotSpot: a compact thermal modeling method for CMOS VLSI systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501-513, May 2006.

[36] W. Hung, G. M. Link, Y. Xie, V. Narayanan, and M. J. Irwin, "Interconnect and thermal-aware floorplanning for 3D microprocessors," *the 7th ISQED*, pp. 98-104, 2006.

[37] C. Isci, M. Martonosi, "Runtime power monitoring in high-end processors: methodology and empirical data," *the 36th Annual International Symposium on Microarchitecture,* pp. 93-104, 2003.

[38] R. Joseph, M. Martonosi, "Run-time power estimation in high-performance microprocessors," *International Symposium on Low Power Electronics and Design, ISLPED,* pp. 135-140, 2001.

[39] J. Joyner, P. Zarkesh-Ha, and J. Meindl, "A stochastic global net-length distribution for a three-dimensional system on chip (3D-SoC)," *the 14th IEEE International ASIC/SOC Conference,* 2001.

[40] W. Kim, M. S. Gupta, G.-Y. Wei, D. Brooks, "System level analysis of fast, per-core DVFS using on-chip switching regulators," *HPCA,* 2008: 123-134.

[41] 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.

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

[43] Eren Kursun, Chen-Yong Cher, "Variation-aware thermal characterization and management of multi-core architectures," ICCD 2008: 280-285.

[44] E. Kursun, C. Y. Cher, A. Buyuktosunoglu, P. Bose, "Investigating the effects of task scheduling on thermal behavior", *the 3rd Workshop on Temperature-Aware Computer Systems, Held in conjunction with ISCA-33*, 2006.

[45] Y. Li, B. Lee, D. Brooks, Z. Hu, K. Skadron, "CMP design space exploration subject to physical constraints" *the 12th IEEE International Symposium on High Performance Computer Architecture (HPCA),* Feb, 2006

[46] Y. Li, D. Brooks, Z. Hu, K. Skadron, "Performance, energy, and thermal considerations for SMT and CMP architectures," *the 11th International Symposium on High-Performance Computer Architecture,* pp. 71-82, 2005.

[47] Gabriel H. Loh, "3D-stacked memory architectures for multi-core processors", ISCA 2008 453-464.

[48] G. H. Loh, Y. Xie, B. Black, "Processor design in 3D die-stacking technologies," *IEEE Micro*, 27(3):31-48, 2007.

[49] R. McGowen, "Adaptive designs for power and thermal optimization," *International Conference on Computer Aided Design,* pp. 118-121, 2005.

[50] Francisco J. Mesa-Martinez, Ehsan K. Ardestani, Jose Renau "Characterizing processor thermal behavior," *ASPLOS* 2010: 193-204.

[51] P. C. Monferrer, G. Magklis, J. González, A. González, "Distributing the frontend for temperature reduction," *the 11th International Symposium on High-Performance Computer Architecture,* pp. 61-70, 2005.

[52] 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.

[53] N. Muralimanohar, R. Balasubramonian, N. P. Jouppi "Architecting Efficient Interconnects for Large Caches with CACTI 6.0," *IEEE Micro* 28(1): 69-79 (2008).

[54] S. Mysore, B. Agrawal, N. Srivastava, S. Lin, K. Banerjee, T. Sherwood, "Introspective 3D chips," *ASPLOS 2006*, pp. 264-273.

[55] P. Pillai, K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," *The 18th ACM Symposium on Operating Systems Principles,* pp. 89-102, 2001.

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

[57] K. Puttaswamy, G. H. Loh "Thermal analysis of a 3D die-stacked high-performance microprocessor," *ACM Great Lakes Symposium on VLSI*, pp. 19-24, 2006.

[58] K. Puttaswamy, G. H. Loh, "Thermal herding: microarchitecture techniques for controlling hotspots in high-performance 3D-integrated processors," *HPCA*, pp. 193-204, 2007.

[59] S. R. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, J. Torrellas, "VARIUS: A model of process variation and resulting timing errors for microarchitects", IEEE Transactions on Semiconductor Manufacturing, Volume 21, Issue 1, Feb. 2008 Page(s):3 - 13

[60] K. Skadron, T. Abdelzaher, M. R. Stan, "Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management," *the 8th International Symposium on High-Performance Computer Architecture,* pp. 17-28, 2002.

[61] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, D. Tarjan, "Temperature-aware microarchitecture," *the 30th International Symposium on Computer Architecture,* pp. 2-13, 2003.

[62] K. Skadron, K. Sankaranarayanan, S. Velusamy, D. Tarjan, M. R. Stan, and W. Huang., "Temperature-aware microarchitecture: modeling and implementation," *ACM TACO*, 1(1):94-125, Mar. 2004.

[63] J. Srinivasan, S. V. Adve, "Predictive dynamic thermal management for multimedia applications," *the 17th Annual International Conference on Supercomputing,* pp. 109-120, 2003.

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

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

[66] Chong Sun, Li Shang, Robert P. Dick, "Three-dimensional multiprocessor system-on-chip thermal optimization," CODES+ISSS 2007: 117-122.

[67] J. Dorsey, S. Searles, M. Ciraula, E. Fang, S. Johnson, N. Bujanos, R. Kumar, D.Wu, M. Braganza, and S. Meyers, "An integrated quad-core Opteron processor," *ISSCC 07:IEEE International Solid-State Circuits Conference Digest of Technical Papers*, 2007.

[68] Radu Teodorescu, Josep Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," ISCA 2008: 363-374.

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

[70] Yefu Wang, Kai Ma, Xiaorui Wang, "Temperature-constrained power control for chip multiprocessors with online model estimation, " ISCA 2009: 314-324.

[71] S. Wang, R. Bettati, "Delay analysis in temperature-constrained hard real-time systems with general task arrivals," *IEEE Real-Time Systems Symposium,* pp. 323-334, 2006.

[72] D. West, "Introduction to Graph Theory (2nd ed.)," *Prentice Hall*, Chapter 3, 1999.

[73] W. Wu, L. Jin, J. Yang, P. Liu, S. X. Tan "A systematic method for functional unit power estimation in microprocessors," *DAC 2006*, pp. 554-557.

[74] W. Wu, L. Jin, J. Yang, P. Liu, S. X.Tan "Efficient power modeling and software thermal sensing for runtime temperature monitoring," *ACM Trans. Design Autom. Electr. Syst.* 12(3): (2007).

[75] Y. Xie, G. Loh, B. Black, and K. Bernstein, "Design space exploration for 3D architecture," *ACM Journal of Emerging Technologies for Computer Systems*, Vol. 2. No. 2, pp. 65-103, April 2006.

[76] J. Yang, X. Zhou, M. Chrobak, Y. Zhang, L. Jin, "Dynamic thermal management through task scheduling," *ISPASS,* pp. 191-201, 2008.

[77] W. Yuan, K. Nahrstedt, "Energy-efficient soft real-time CPU scheduling for mobile multimedia systems," *the 19th ACM Symposium on Operating Systems Principles,* pp. 149-163, 2003.

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

[79] X. Zhou, Y. Xu, Y. Du, Y. Zhang, J. Yang, "Thermal management for 3D processor via task scheduling," *International Conference on Parallel Processing,* Sept. 2008.

[80] C. Zhu, Z. P. Gu, L. Shang, R. P. Dick, and R. Joseph, "Three-dimensional chip-multiprocessor run-time thermal management," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, no. 8, Aug. 2008.

[81] http://www.hardocp.com/article/2010/01/03/intel_westmere_32nm_clarkdale_core_i5661_review/

[82] http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html

[83] http://en.wikipedia.org/wiki/Dynamic_frequency_scaling

[84] http://en.wikipedia.org/wiki/Dynamic_Voltage_Scaling

[85] "Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor", white paper, Intel, March, 2004, http://xnu-speedstep.googlecode.com/files/PentiumM_SpeedStepDoc.pdf.

[86] "AMD PowerNow! Technology Platform Design Guide for Embedded Processors Application", application note, AMD, December, 2000, http://www.amd.com/epd/processors/6.32bitproc/8.amdk6fami/x24267/24267a.pdf.

[87] "Intel Pentium 4 processor in the 478-pin package thermal design guidelines," design guide, Intel, May 2002, http://developer.intel.com/design/pentium4/guides/249889.htm

[88] "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A," *http://www.intel.com/Assets/PDF/manual/253668.pdf.*

[89] http://en.wikipedia.org/wiki/List_of_Intel_microprocessors.

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

[91] http://www.spec.org/cpu2006/