ThermOS: System Support for Dynamic Thermal Management of Chip Multi-Processors

Filippo Sironi*, Martina Maggio†, Riccardo Cattaneo*, Giovanni F. Del Nero*
Donatella Sciuto*, Marco D. Santambrogio*
*Politecnico di Milano, Milano, Italy, †Lund University, Lund, Sweden
{sironi, rcattaneo, sciuto, santambrogio}@elet.polimi.it, martina.maggio@control.lth.se, giovanni.delnero@mail.polimi.it

Abstract—**Constraining the temperature of computing systems has become a dominant aspect in the design of integrated circuits. The supply voltage decrease has lost its pace even though the feature size is shrinking constantly. This results in an increased number of transistors per unit of area and hence a growing power density. Researchers started investigating dynamic thermal management techniques to address the trade-off between performance and temperature. Hardware dynamic thermal management can guarantee safety but, at the same time, can negatively affect established service-level agreements. On the other hand, software solutions rely on hardware for safety but does not indiscriminately trade-off performance for temperature.**

We propose *ThermOS*, an extension for commodity operating systems that harnesses formal feedback control and idle cycle injection to decrease thermal emergencies while showing better efficiency than commodity and cutting edge techniques.

Index Terms—**dynamic thermal management, DTM, chip multi-processors, CMP, multi-core, operating systems, OS**

I. Introduction

The shift from single-core superscalar processors to multi-core processors was a tentative response to address the inability of respecting Joy's law: the peak computer speed doubles each year.[1] If parallel software is available, a multi-core processor made up of smaller cores, which harness thread-level parallelism, can outperform a massive single-core superscalar processor exploiting instruction-level parallelism within the same power budget.

In the last decade we assisted to the proliferation of multi-core processors such as chip multi-processors (CMPs) and multi-processors system-on-chip (MPSoCs) characterized by a constantly increasing number of transistors made possible by the ever-decreasing feature size. However, recent lithographic technologies do not abide Dennard's scaling law causing power density of a multi-core processors to approach that of a nuclear reactor. Power density increases as the scaling of clock frequency and number of transistors outpaces the downscaling of supply voltage. The consequent rise of temperature due to the inability of packages to dissipate heat heavily influences the design of computing systems.

Maintaining temperature under control is crucial for performance, energy consumption, and reliability of integrated circuits: a higher temperature increases leakage current and leads to a sharp increase of energy consumption [1] and to drastic decreases of both throughput [2] and mean time to failure (MTTF) [3]. Researchers from the computer architecture, compiler, and operating system communities put efforts in addressing this issue. Our work pursues the same objective.

We propose *ThermOS* (Thermal Operating System), an extension for commodity operating systems, which provides dynamic thermal management (DTM) through formal feedback control and idle cycle injection (ICI) [4] for multi-programmed workloads. *ThermOS* specifically targets commodity CMPs, which cannot benefit from the latest architectural and micro-architectural advancements. However, we believe that *ThermOS* could benefit even further from both the architectural and micro-architectural evolution.

This paper makes the following contributions. (1) Propose and validate a linear discrete-time thermal model that describes the temperature behavior around the threshold when employing ICI for DTM. (2) Derive a proportional-integral (PI) controller to drive ICI, demonstrate its asymptotic stability and robustness. (3) Evaluate *ThermOS* on a commodity CMP with representative benchmarks showing its capabilities of managing multi-programmed workloads and addressing the trade-off between temperature and performance.

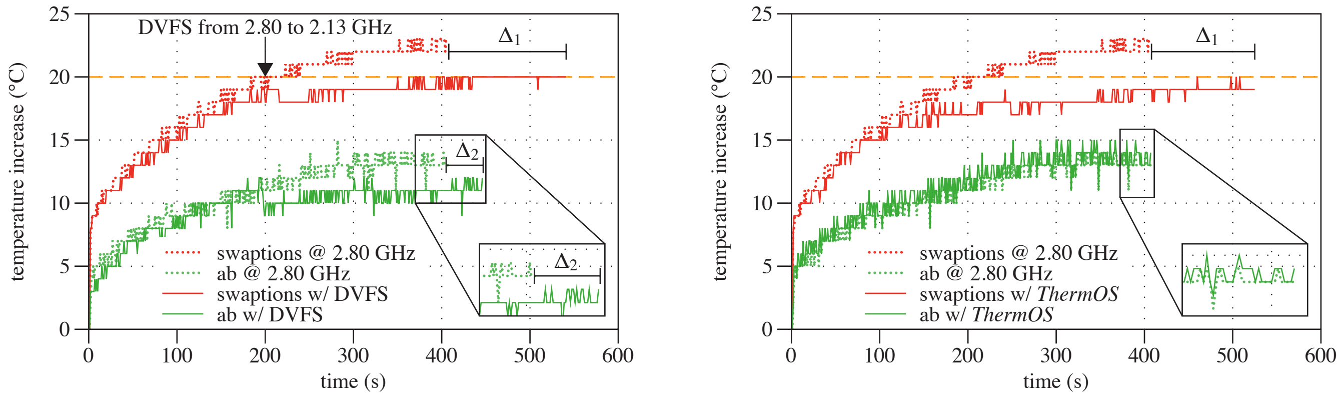
The remainder of this paper is organized as follows. Section II makes a first high-level comparison between DVFS and *ThermOS*. Section III describes the linear discrete-time thermal model that enables *ThermOS* while Section IV reports implementation details regarding each *ThermOS* component. Section V provides evidence that *ThermOS* achieves its goals. Section VI surveys, at the best of our knowledge, related work and highlights benefits and drawbacks of *ThermOS* with respect to the state of art. Finally, Section VII concludes the paper.

II. Dynamic Thermal Management

Typical scheduling algorithms implement the race to idle approach: applications run as fast as possible to allow processors entering low power states as soon as possible. This behavior leverages the capability of decreasing energy consumption when employing low power states and delivers the best performance. Race to idle favors energy efficiency [5] and is beneficial for desktops, laptops, and mobiles, where interactive, low-utilization applications are common.

Conversely, race to idle leads to high temperature in servers and large-scale computing systems where non-interactive high-utilization applications prevail, incurring in additional costs to power computer room air conditioning (CRAC) and heat, ventilation, and air conditioning (HVAC). CRAC and HVAC

[1]J. Markoff, "The not-so-distant future of personal computing," InfoWorld, no. 49, 1993.



(a) DVFS. (b) *ThermOS*.

Fig. 1. Executions of a multi-programmed workload on a commodity CMP. On the left: DVFS decreases temperature with a chip-wide impact on performance. On the right: *ThermOS* decreases temperature with a core-wide impact on performance.

are in place to avoid exceeding the temperature threshold[2] and limit the number of DTM events that degrade vital measures such as throughput, latency, and missed deadlines.

A. Dynamic Voltage and Frequency Scaling

Researchers from the computer architecture community demonstrated the energy efficiency of per-core P-states [6] through DVFS in CMPs [7]. Each core within a CMP heats up differently depending on the manufacturing variability of the silicon, the floorplan, the application it is running, etc. [1] making the adoption of per-core P-states desirable to address temperature issues. However, providing such a fine-grained control in CMPs with more than few cores is uncommon [8] and most of the manufacturers ditch fine-grained control in favor of coarse-grained control.[3]

This scenario is especially harmful for multi-tenant environments. Each user is assigned a certain amount of resources and expects predictable performance. The occurrence of DTM events (e.g., the activation of DVFS as a consequence of the overheating of a single core running a particularly CPU-intensive application) affects the whole CMP and impairs the performance of all the applications within a multi-programmed workload, regardless of the owner.

Fig. 1a depicts this setting. We run *swaptions*, a CPU-intensive application from the PARSEC 2.1 benchmark suite [9], and *apachebench*, an I/O-intensive application, on two different cores. When running at the highest clock frequency, the core executing *swaptions* overheats (see the dashed red line in Fig. 1a), breaking the temperature threshold, while the core executing *apachebench* does not (see the dashed green line in Fig. 1a). When the same multi-programmed workload executes on a CMP using chip-wide DVFS for DTM, each core is subject to the same supply voltage and clock frequency setting. The core executing *swaptions* does not overheat when DVFS is in place since it decreases the supply voltage and the clock frequency (see the point at 200 s in the execution in Fig. 1a); this directly translates into a run time increase (see red line and Δ_1 in Fig. 1a). Unfortunately, the same happens—on a smaller scale due to the I/O-intensive nature of the application—to *apachebench*, which does not cause overheating at the highest clock frequency (see the light green line). Hence, *apachebench* is unnecessarily slowed down (see green line and Δ_2 in Fig. 1a).

[2]The temperature threshold can be either a manufacturer-defined safety limit or an administrator-defined cap to lower the total cost of ownership.

[3]Commodity CMPs support per-core P-states; unfortunately, this setting becomes effective only when cores operates in different C-states.

B. Idle Cycle Injection

The system-wide performance degradation of chip-wide DVFS is its main drawback. *ThermOS* addresses this issue by harnessing formal feedback control and ICI [4]. Let us reconsider the previous multi-tenant scenario. *ThermOS* selectively throttles the execution of those applications whose cores are overheating without affecting the remaining cores and thus avoiding system-wide performance degradation. Fig. 1b depicts this setting. *ThermOS* prevents the core running *swaptions* from overheating by increasing its run time (see red line and Δ_1 in Fig. 1b). At the same time, it does not impact the execution of *apachebench* (see green lines in Fig. 1b).

Almost all processors today support a handful of C-states—five on Intel “Ivy Bridge”—and this trend is spreading as processors dynamic power and thermal management gain momentum [10]. For example, our evaluation platform features an Intel Xeon Processor W3530 supporting the C0, C1/C1E, C3, and C6 states at the individual thread, core, and package level. A convenient interface accessible through the `MWAIT` instruction allows the operating system requesting low power states [11]. *ThermOS* exploits this interface to selectively throttle applications, thus decreasing temperature.

Flexibility makes ICI very interesting. For example, Google is already exploiting ICI through *kidled* [12] in some of its data centers while Intel recently merged *PowerClamp* [13]—a thermal driver for ICI— with the Linux kernel 3.9 (released April 29, 2013). Other approaches, like *Dimetrodon* [4], rely on ICI to provide preventive thermal management via probabilistic injection of idle time. We thoroughly compares *Dimetrodon* with *ThermOS* in Section V.

III. Thermal Model

DTM can benefit from accurate thermal modeling and many proposals can be found in the computer architecture literature. Brooks and Martonosi [14] model the temperature behavior through power consumption in the *Wattch* power analysis framework. Unfortunately, chip-wide power consumption is a poor proxy for temperature [1].

Skadron et al. [1] model the temperature behavior through a compact thermal model in the *HotSpot* thermal analysis framework. This solution is fairly accurate and explains the complete temperature behavior. The main disadvantage of the compact thermal model is the need for a considerable amount of micro-architectural details such as the floorplan of the functional units. This information may be available for obsolete designs but can only be guessed for current ones.

Zhou et al. [15] harnesses the compact thermal model to deploy a thermal-aware scheduler, which requires the complete temperature behavior for minimization without hurting performance. However, the compact thermal model is simplified to make its adoption viable outside of a simulation environment and inside the Linux kernel.

Commodity designs cannot benefit from the latest advancements in micro-architectural DTM [16]. They rely on conservative techniques like DVFS to guarantee safety. Given safety for granted, portable software DTM techniques like ICI become attractive to address the trade-off between temperature and performance, as shown in Section II.

A. *Thermal Model for Dynamic Thermal Management*

Deriving a thermal model that is meaningful for the whole range of commodity designs and DTM techniques is impractical. Thus, we propose one that assumes the availability of software DTM and explains the temperature behavior around the threshold.

We employ the linear discrete-time thermal model described in Eq. (1). $T(k)$ and $T(k+1)$ are the temperatures at the k-th and $k+1$-th sample instants, respectively; $I(k)$ is the idle time between the k-th and $k+1$-th sample instants; while a and b are parameters defining the temperature behavior.

$$T(k+1) = a \cdot T(k) + b \cdot I(k) \tag{1}$$

According to the linear discrete-time thermal model, we can approximate the future temperature by accounting for its current value and the idle time between the current and future time instants.

Fig. 2 shows a thermal simulation leveraging the compact thermal model proposed by Skadron et al. [1]. We simulate a worst-case application capable of pushing temperature of an "abstract" single-core processor up to $80\,^\circ\mathrm{C}$ given an idle temperature of $30\,^\circ\mathrm{C}$ (see the red line labeled "w/o ICI" in Fig. 2). One should note that the temperature behavior has been artificially accelerated to show a meaningful time frame.

The thermal simulation comprises ICI for DTM to alternate high energy consumption phases with low energy consumption phases. The control period for ICI is set to $10\,\mathrm{ms}$ (see the ticks on the blue dashed line in Fig. 2) while the idle time can

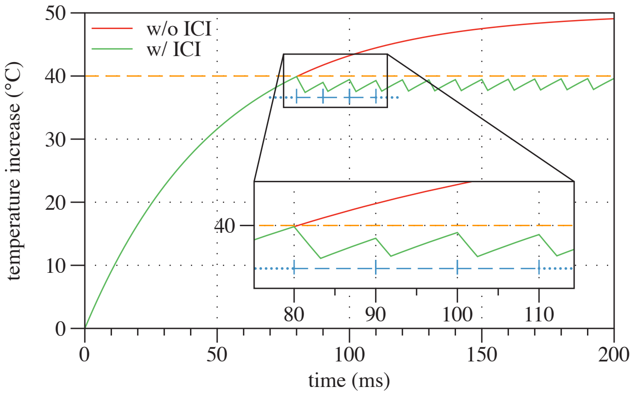


Fig. 2. Thermal simulation leveraging the compact thermal model giving visual evidence on the applicability of the linear discrete-time thermal model.

reach at most $80\,\%$ of this value (i.e., $8\,\mathrm{ms}$). The temperature threshold is set to $70\,^\circ\mathrm{C}$ while the trigger threshold is set to $1\,^\circ\mathrm{C}$ less. On the first thermal emergency at $80\,\mathrm{ms}$, *ThermOS* injects $\approx$ $30\,\%$ of idle time and, from that point on, it injects $\approx$ $20\,\%$ of idle time per control period to keep temperature around the threshold (see the green line in Fig. 2).

The simulation gives visual evidence that the temperature behavior around the threshold when employing ICI for DTM is quasi-linear both when ICI does and does not inject idle time and provides a first hint on the applicability of our thermal modeling approach.

B. *Thermal Model Training*

The estimation of parameters can be performed either online or offline and combinations of the two may apply. Online estimation is beneficial for time-variant workloads alternating CPU with memory and I/O-intensive phases and for workloads with CPU-intensive phases in which the number of instructions issued per clock cycle has a high variance, since it allows a thermal model to better track the temperature behavior. However, online estimation introduces overhead at run time since usually the better the estimation algorithm the higher its execution time. Offline estimation is suited for steady time-invariant workloads characterized by a single phase that is either CPU, memory, or I/O-intensive. Since the estimation of parameters occurs offline, the run time overhead is completely absent. Offline estimation requires an accurate training phase to guarantee that a thermal model fits the temperature behavior.

We decided to use offline estimation for the following reasons: (1) we focus on multi-programmed CPU-intensive workloads that have a high probability of increasing temperature; (2) we use a reasonably high control frequency in the realm of operating systems and hence we must keep the temporal overhead under control; and (3) we execute in kernel-mode and hence we are discouraged from using floating point computation; this makes almost prohibitive the implementation of most estimation algorithms at run time.

C. *Estimation and Empirical Validation*

We run a modified version of *ThermOS* that randomly selects a value for the idle time in the interval $[0\,\%, 80\,\%]$ on our evaluation platform. We run a worst-case workload consisting of four instances of *cpuburn* to make the linear discrete-time thermal model conservative with respect to real-world workloads. We collected about 1.5 millions of triples—equivalent to $\approx 1\,\mathrm{h}$ of execution—consisting of the current temperature value, the previous temperature value, and the idle time value to catch most of the temperature behavior.

We used the least squares algorithm to estimate the parameters and fit the linear discrete-time thermal model reported in Eq. (1). The least squares algorithm "solves" the linear system $y = X \cdot w$, where y is the column vector made up of n values of the current temperature, X is the matrix of n tuples consisting of the previous temperature value and the idle time value, and w is the column vector containing the m parameters (i.e., a and b).

We partitioned the collection of triples in two: a training set of $70\,\%$ of the triples and a validation set of $30\,\%$ of the triples. We run the algorithm on the training set and verified the result against the validation set. The trained linear discrete-time thermal model yields a correct prediction for over $95\,\%$ of the triples of the validation set. Although we expect lower accuracy with real-world workloads due to the conservative nature of our thermal modeling approach, it is still accurate enough. We iterated the estimation many times with different training and validation sets to gain information about the robustness of parameters estimation. We eventually selected the best couple of parameters where a and b are 1.0244 and -0.0484, respectively.

While the simulation of *ThermOS* shows that the linear discrete-time thermal model is meaningful, the empirical validation gives mathematical evidence and strengthen our thermal modeling approach.

D. Statistical Validation

We further evaluated the quality of the estimated parameters values through the computation of their statistical significance. Since we used the least squares algorithm, it was possible to estimate the variance of the parameters by means of the statistic reported in Eq. (2). w_i is the i-th parameter; X is the matrix of coefficient of the linear system "solved" by the least squares algorithm; S is the sum of squared residuals computed according to Eq. (3); n and m are the number of triples used by the least squares algorithm and the number of parameters of the linear discrete-time thermal model, respectively.

$$\mathrm{Var}(w_i) = \sigma^2([X^T \cdot X]^{-1})_{ii} \approx \frac{S}{n-m} \cdot ([X^T \cdot X]^{-1})_{ii} \quad (2)$$

$$S = \sum_{i=1}^{n} (y_i - X_i \cdot w)^2 \quad (3)$$

The estimated variances of the a and b parameters values across different training/validation partitions are $6.7851 \cdot 10^{-8}$ and $1.4059 \cdot 10^{-7}$, respectively. They suggest our thermal modeling approach is robust.

IV. Thermal Manager

We structured *ThermOS* following a feedback design and implemented it inside the Linux kernel. The first step consists in observing temperature values. The second step takes decisions regarding the needed idle time. Finally, the third step incorporates the idle time into applications execution.

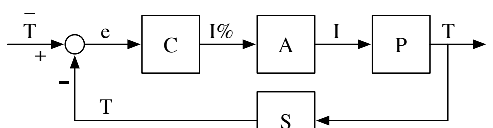


Fig. 3. Feedback design.

A. Temperature Measurement

Formal feedback control requires contextual information; more specifically, DTM leverages temperature "measurements". One can either rely on analytic thermal models or employ thermal sensors to provide such "measurements".

ThermOS provides DTM through a software solution that mitigates the drawbacks of hardware DTM. However, software DTM cannot provide strong guarantees of limiting temperature; thankfully, *ThermOS* can still rely on hardware DTM to face thermal emergencies. Because of this reason, the low resolution of temperature measurements obtained through on-chip digital thermal sensors (DTSs) available on commodity CMPs are sufficient. For example, our evaluation platform features an Intel Xeon Processor W3530 supporting per-core DTSs with a resolution of $1\,°\mathrm{C}$.

We implemented the observation phase through high-priority kernel-mode threads. Each high-priority kernel-mode thread always executes on the same core and periodically probes machine-specific registers to retrieve the temperature measurement of the core it is running on.

B. Idle Time Determination

Formal feedback control is successful in managing systems explained through mathematical models. The physical laws ruling thermal phenomena provide a strong mathematical model enabling the use of formal feedback control for DTM, thus avoiding the difficulties associated with specially-designed controllers.

Formal feedback control provides many advantages thanks to its formalism. It is possible to design controllers with predictable behavior in terms of response time and to achieve desirable stability and robustness guarantees.

Control theory helps synthesizing controllers that achieve the desired output by exploiting the availability of mathematical models of the controlled processes. In particular, industry strongly relies on formal feedback control and harnesses well-known solutions that proved beneficial even when dealing with approximate mathematical models: P, PI, and PID (proportional-integral-derivative) controllers.

1) Formal Feedback Controller: Fig. 3 shows a feedback design, where the sensor S measures the output T of the process or plant P; the controller C computes the error e subtracting the output T from the desired output $\bar{T}$ and then constraints process the input I through the actuator A. Following this paradigm, we realize per-core formal feedback controllers.

Let T_{emerg} be the temperature threshold such that exceeding such value causes a thermal emergency. We must start DTM before reaching T_{emerg}, so we set a temperature trigger threshold $\bar{T} < T_{\text{emerg}}$. We periodically sample the temperature measurement $T_i(k)$ of core i following the methodology reported in Section IV-A. We compute the error $e_i(k) = \bar{T} - T_i(k)$. If the error $e_i(k)$ is negative the i-th core is overheating. Otherwise, if the error $e_i(k)$ is positive, the i-th core is working properly.

We devised a PI controller that responds to errors by means of two terms: (1) a proportional term and (2) an integral term. The proportional term changes its effect according to the current value of the error and in a way that decreases the future values of the error. The integral term changes its effect incrementally accounting for the past values of the error. We neglected the derivative term; while this results into a little loss of control, at the same time it leads to notably less noise.

The synthesis of the PI controller depends on whether the mathematical model of the process is continuous-time or discrete-time. Since we periodically obtain per-core temperature measurements with a specified sample period, the mathematical model reported in Eq. (1) is discrete-time. Thus, we perform the derivation in the $\mathcal{Z}$-transform domain.

Eq. (4) represents the PI controller for core i, where $I_i(k)$ is the idle time required to constrain temperature, expressed as a percentage of the control period.

$$I_i(k) = I_i(k-1) + \frac{1-p}{b} \cdot e_i(k) - a \cdot \frac{1-p}{b} \cdot e_i(k-1) \quad (4)$$

2) Controller Synthesis and Stability Proof: We devised a PI controller for the linear discrete-time thermal model reported in Eq. (1) with the goal of keeping temperature $T(k)$ as close as possible to the trigger threshold $\bar{T}$.

We determined the transfer function $\mathcal{P}(z)$ applying the $\mathcal{Z}$-transform to the linear discrete-time thermal model reported in Eq. (1). Eq. (5) shows the result, where $\mathcal{T}(z)$ and $\mathcal{I}(z)$ are the $\mathcal{Z}$-transforms of temperature and idle time, respectively.

$$\begin{aligned} z \cdot \mathcal{T}(z) &= a \cdot \mathcal{T}(z) + b \cdot \mathcal{I}(z) \\ \mathcal{P}(z) &= \frac{\mathcal{T}(z)}{\mathcal{I}(z)} = \frac{b}{z-a} \end{aligned} \quad (5)$$

We synthesized the PI controller by constraining the transfer function of the feedback as explained by Levine [17]. Eq. (6) holds the result; $\mathcal{G}(z)$ and $\mathcal{C}(z)$ are the transfer functions of the feedback and of the PI controller, respectively.

$$\mathcal{G}(z) = \frac{\mathcal{C}(z) \cdot \mathcal{P}(z)}{1 + \mathcal{C}(z) \cdot \mathcal{P}(z)} = \frac{1-p}{z-p} \quad (6)$$

We employed a first order transfer function with a pole in p, a configurable parameter whose value changes the responsiveness of the PI controller. If p is chosen in the interval $(-1, 1)$ the feedback is asymptotically stable. Moreover, if p is chosen in the interval $(0, 1)$ the feedback guarantees the absence of oscillations. Given asymptotic stability and the absence of oscillations for granted, large values of p in the interval $(0, 1)$ translate into a slower but smoother response, while small values of p translate into a faster but rougher response. *ThermOS* provides a compile time setting to change the value of p in the interval $(0, 1)$, therefore we conclude the feedback is asymptotically stable.

Starting from Eq. (6), we determined the transfer function $\mathcal{C}(z)$ of the controller; Eq. (7) holds the result.

$$\mathcal{C}(z) = \frac{(1-p) \cdot (z-a)}{b \cdot (z-1)} \quad (7)$$

We imposed $\mathcal{C}(z) = \mathcal{I}(z)/\mathcal{E}(z)$; this leads to the transfer function reported in Eq. (8).

$$\begin{aligned} \frac{\mathcal{I}(z)}{\mathcal{E}(z)} &= \frac{(1-p) \cdot (z-a)}{b \cdot (z-1)} \\ z \cdot \mathcal{I}(z) - \mathcal{I}(z) &= z \cdot \frac{1-p}{b} \cdot \mathcal{E}(z) - a \cdot \frac{1-p}{b} \cdot \mathcal{E}(z) \end{aligned} \quad (8)$$

The $\mathcal{Z}$-antitransform and a time shift applied to Eq. (8) yield Eq. (9), the generic form of Eq. (4).

$$I(k) = I(k-1) + \frac{1-p}{b} \cdot e(k) - a \cdot \frac{1-p}{b} \cdot e(k-1) \quad (9)$$

3) Controller Robustness Analysis: In general, a controller depends on the process it is responsible for. In our case, the PI controller depends on the linear discrete-time thermal model depicted in Eq. (1), whose parameters were estimated and validated in Section III-B.

The statistical significance of the a and b parameters values makes us confident. However, *ThermOS* must deal with many events that can suspend ICI; this negatively impacts the effectiveness of ICI, which is represented by the b parameter. For this reason, we ask ourselves: what if our thermal modeling approach is not faithful and, in particular, what if the b parameter of the linear discrete-time thermal model is poorly estimated? In other terms, what if the weight of $I(k)$ is not b but $b + \delta$? We answer this question by means of a robustness analysis.

We assume the real process is described by the transfer function $\mathcal{P}(z)$ reported in Eq. (10).

$$\mathcal{P}(z) = \frac{b+\delta}{z-a} \quad (10)$$

We substitute Eq. (10) in Eq. (6). The pole of the transfer function of the feedback changes from $z = p$ to Eq. (11).

$$z = \frac{p \cdot (b+\delta) - \delta}{b} \quad (11)$$

If the pole lays in the interval $(-1, 1)$ the feedback remains asymptotically stable and loses at most the guarantee on the absence of oscillations. We solve the system of inequalities that leads to Eq. (12).

$$\delta > \frac{b \cdot (1+p)}{1-p} \quad \wedge \quad \delta < -b \quad (12)$$

In practice, δ can vary in the interval $(-0.0899, 0.0484)$ when the b and p parameters values are -0.0484 and 0.3, respectively. The interval is large when compared to the estimated value of the b parameter. Hence, we declare that *ThermOS* is robust with respect to estimation errors on the effectiveness of ICI.

C. Idle Cycles Injection

The scheduling infrastructure of the Linux kernel enables different algorithms to schedule different types of threads (i.e., either a process or a thread). This materializes in different scheduling classes with different priorities. The scheduling skeleton iterates over scheduling classes from the highest to the lowest priority to pick the next runnable thread.

The scheduling infrastructure of the Linux kernel provides five scheduling classes: (1) `SCHED_FIFO`, (2) `SCHED_RR`, (3) `SCHED_OTHER`, (4) `SCHED_BATCH`, and (5) `SCHED_IDLE`. The first two scheduling classes provide "real-time" policies while the remaining provide "normal" policies.

We implemented ICI for user-mode threads scheduled through normal policies, which are under the control of the Completely Fair Scheduler (CFS). This is consistent with previous implementations of ICI [4]. The rationale behind this choice is avoiding the preemption of real-time threads, which are rarely present in most GNU/Linux systems, and kernel-mode threads, which usually run with low IPC, causing low power consumption and temperature.

When the scheduling skeleton calls CFS, *ThermOS* enters the actuation phase and may or may not perform ICI depending on the outcome of the decision phase. We implemented ICI within the Linux kernel exploiting the availability of an idle thread for each core. CFS eventually picks the idle thread instead of the next runnable thread and runs it for as long as the thermal controller (i.e., the PI controller) dictates.

1) Changing the Idle Task: The Linux kernel executes the idle thread whenever there are no runnable threads; the idle thread yields as soon as a thread becomes runnable. *ThermOS* also executes the idle thread whenever the thermal controller demands the injection of idle time.

Without loss of generality, we analyze the behavior of the Linux kernel when executing on top of Intel x86 and x86-64 processors. The Linux kernel runs the idle thread in kernel-mode to allow the execution of protected instructions. The idle thread issues the `MONITOR` instruction to arm the address monitoring hardware with the address of the `flags` variable stored in its `task_struct`. It then issues the `MWAIT` instruction to request the processor entering the C1E state.

The Linux kernel eventually writes the `flags` variable of the idle thread to demand a reschedule. The address monitoring hardware catches the write operation forcing the processor to exit the C1E state and enter the C0 state. Finally, the `MWAIT` instruction returns and the idle thread yields.

We modified the idle thread to issue the `MONITOR` instruction to arm the address monitoring hardware with different variables depending on the outcome of the thermal controller. Whenever the thermal controller demands the injection of idle time, the idle thread selects the `thermal_flags` variable, which is once again stored in its `task_struct`. Otherwise, the idle thread selects the `flags` variable and its behavior is unmodified. The idle thread then issues the `MWAIT` instruction to request the processor entering the C1E state.

The Linux kernel eventually writes either the `thermal_flags` or the `flags` variable of the idle thread to indicate the idle time is exhausted or a runnable thread is available possibly triggering a C-state transition. In addition, the Linux kernel writes the `thermal_flags` instead of the `flags` variable of the idle thread whenever the idle thread is running for cooling purpose and either a real-time or a kernel-mode thread became runnable. This grants *ThermOS* with the capability of suspending ICI to face the execution of real-time or kernel-mode threads.

2) Exploiting the Dynamic Tick: The Linux kernel uses a periodic timer firing at a configurable frequency—100, 250, 333, 1000 Hz—for "house-keeping" operations. This timer is usually referred to as *scheduler tick*.

The scheduler tick forces the processor to exit low power states and hence increases the energy consumption even when the Linux kernel is executing the idle thread. Since energy consumption is a fundamental issue, the Linux kernel 2.6.21 introduced the *dynamic scheduler tick*. The scheduler tick is temporarily disabled to "idle" instead of "idle with ticks". The scheduler tick fires periodically whenever the Linux kernel executes runnable threads while it fires on-demand whenever the Linux kernel executes the idle thread.

When the Linux kernel sets the scheduler tick to fire on-demand it takes the difference between the current time and the next time a software interrupt request must execute. We modified this behavior by choosing the minimum between the next time a scheduled interrupt request (e.g., `sleep(2)`) must execute and the idle time.

3) Scheduling the Idle Task and Alternatives: In the remainder of this section we analyze alternative approaches to scheduling the idle thread as a means for ICI and we highlight the choices that led to our design.

Scheduling the idle thread as a means for ICI may be sub-optimal from a performance standpoint: it requires *trapping* from user to kernel-mode and *context switching* the current thread with the idle thread. A first alternative approach targets the context switching issue. One could avoid the cost of context switching from the current thread to the idle thread by "replicating" the functionality of the idle thread and the *cpuidle* infrastructure: issue the `MONITOR` and `MWAIT` protected instructions that allow the processor to arm the address monitoring hardware and transition from the C0 to the C1E state. This approach violates the basics of software engineering by replicating a well-structured functionality.

A second alternative approach targets both the trapping and the context switching issues. One could avoid the cost of trapping from user to kernel-mode and hence the cost of context switching from the current thread to the idle thread by issuing "low-power" instructions in user-mode. The adoption of a just-in-time (JIT) compiler allows changing the code of an application at run time; it is theoretically possible harnessing this feature to "inject" a series of `NOP` instructions to cool down the processor. Unfortunately, the effectiveness of this approach is limited by design by the use of the NOP instruction, which does not allow the processor to transition from the C0 to the C1E state. We are aware of state-of-the-art compiler-directed techniques to decrease the peak temperature of a processor to improve long-term reliability [18]; however, these techniques target the mitigation of long-term effects like the negative bias temperature instability and the aging.

We quantified the overhead of trapping and context-switching: it is limited between 3 and 30 μs where the worst case accounts for thread migration. We concluded that the overhead is acceptable and does not compromise the efficiency of *ThermOS*.

V. Evaluation

This section evaluates *ThermOS* and, in particular, it is focused on answering the following questions: (1) Can *ThermOS* constrain temperature and affect applications within a multi-programmed workload depending on cores thermal profiles? (2) How efficient is *ThermOS* in tackling the trade-off between performance and temperature when compared to *Dimetrodon* and DVFS?

A. Evaluation Platform and Configuration

We evaluated *ThermOS* on a Dell Precision T3500 workstation with an Intel Xeon Processor W3530 and 12 GB of Single Ranked DIMMs. The processor features 4 cores operating at 2.80 GHz and sharing 8 MB of last-level cache. Each memory module runs at 1066 MHz. The Enhanced Intel SpeedStep Technology allows the processor to work in ten different P-states from 1.60 to 2.80 GHz. We disabled the Intel Turbo Boost Technology to prevent the processor entering P-states with clock frequency higher than the nominal when a subset of the cores is executing. The Intel Turbo Boost Technology would bias the analysis in favor of *ThermOS* that creates unbalanced execution times since it exploits the different thermal profiles. We disabled the Intel Hyper-Threading Technology (HTT) to simplify our implementation. When HTT is enabled, each physical core is in fact a couple of virtual cores requiring ICI co-scheduling to enter the C1E state [11].

We configured Debian 7.0 to run the Linux kernel 3.4 enhanced with *ThermOS* and FreeBSD 7.2 enhanced with *Dimetrodon* [4]. We configured *ThermOS* with a temperature sampling period of 10 ms and a control period of 10 ms. The thermal controller was setup to limit the idle time to 80 % of the control period and the temperature trigger threshold is 3 °C lower than the temperature threshold. The a, b, and p parameters values are 1.0244, −0.0484, and 0.3, respectively. Section III-B supports the choice of these values.

We assessed the behavior of *ThermOS* through the PARSEC 2.1 benchmark suite [9], which provides a set of representative workloads. We ran multi-programmed workloads of single-threaded applications pinned to cores. Section V-D comments on *ThermOS*'s behavior with multi-threaded applications.

TABLE I
Break down of the execution time of a multi-programmed workload per core per C-state

Core	C0	C1E	C3	C6
0	91 %	7 %	2 %	0 %
1	91 %	9 %	0 %	0 %
2	91 %	6 %	3 %	0 %
3	91 %	5 %	4 %	0 %

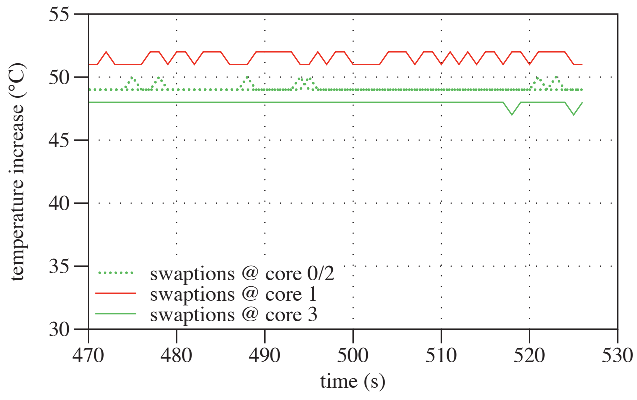
B. Addressing Multi-Programmed Workloads

We first show how *ThermOS* is capable of constraining temperature and affecting applications within a multi-programmed workload depending on cores thermal profiles.

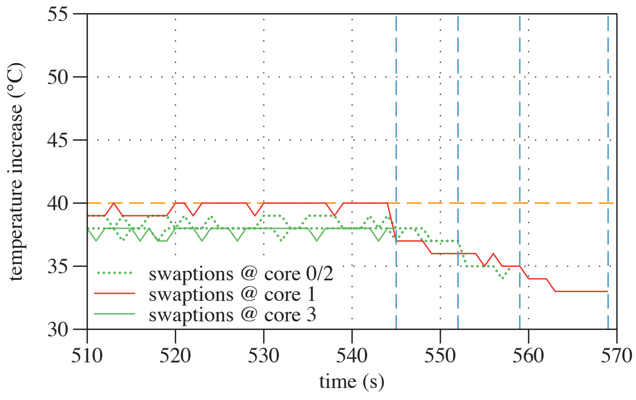
We thoroughly explain the behavior of *ThermOS* when running a homogeneous multi-programmed workload consisting of four instances of *swaptions*, each one running with a single thread of execution on its own core since we believe it helps making our point. However, similar considerations hold for the various multi-programmed workloads we employ in the remainder of this paper.

The multi-programmed workload leads to a steady temperature of about 80 °C with an idle temperature of 30 °C. Fig. 4a shows the last minute of execution without any form of DTM and highlights different thermal profiles for the four cores; core 1 operates at a higher temperature than the other cores and overcomes 80 °C while core 3 operates at a lower temperature. Fig. 4b displays the last minute of execution with *ThermOS* configured to constraint temperature below 70 °C. *ThermOS* enforces the threshold. Table I breaks down the execution time of the multi-programmed workload per core per C-state when executing on *ThermOS*. Each instance of *swaptions* always requires the execution of the same amount of instructions to run to completion and in fact they complete at the same time when any form of DTM is disabled (see Fig. 4a). All cores spend approximately the same percentage of the execution time in C0 since it is the only C-state in which cores actually execute instructions of the instances of *swaptions*.

When executing on *ThermOS*, the real execution time of an instance of *swaptions* accounts for the time spent in C0 and C1E since *ThermOS* exploits only the latter to lower temperature instead of using C3 and C6 that introduce higher latency to enter and exit the C-state (i.e., 20 and 200 μs, respectively, instead of 3 μs) and penalties (e.g., private caches and register file flushes). Cores spend different percentages of the execution time in C1E and C3 since *ThermOS* injects idle time depending on cores thermal profiles and hence the instances of *swaptions* complete at different instants. For example, core 1 operates at a higher temperature than the other cores and the instance of *swaptions* it runs is the last to complete. Thus, core 1 spends the highest percentage of the execution time in C1E when compared to the other cores and none in C3. Conversely, core 3 operates at a lower temperature and the instance of *swaptions* it runs is the first to complete. Thus, core 3 spends the lowest and the highest percentages of the execution time in C1E and C3, respectively, when compared to the other cores.



(a) w/o *ThermOS*.



(b) w/ *ThermOS*.

Fig. 4. Executions of a multi-programmed workload on a commodity CMP. On the left: any form of DTM is disabled, hence temperature rises without constraints. On the right: *ThermOS* is enabled, hence temperature rises but remains constrained below the threshold.

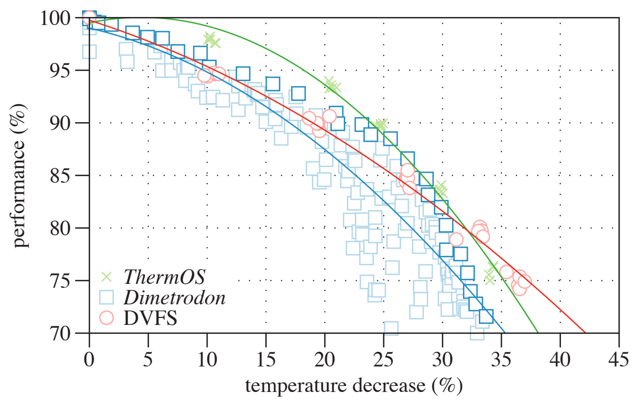


Fig. 5. Efficiency in tackling the trade-off between performance and temperature for *ThermOS*, *Dimetrodon*, and DVFS.

Fig. 4b displays the behavior of *ThermOS* when executing a CPU-intensive workload. However, *ThermOS* is independent from the kind of workload. Fig. 1b highlights the same capabilities even when *ThermOS* executes a workload consisting of a CPU and an I/O-intensive applications.

C. Addressing the Performance/Temperature Trade-Off

We also show how *ThermOS* is efficient in tackling the trade-off between performance and temperature.

We configured *ThermOS* to achieve temperature decreases of: 10, 20, 25, 30, and 35 % with respect to the idle temperature. We configured *Dimetrodon* varying the idle time between 10 and 100 ms and the idle probability between 0 and 40 % for a total of 150 configurations. We statically set the following P-states: 2.79, 2.66, 2.53, 2.39, 2.26, and 2.13 GHz through DVFS. Each P-state is used for the whole execution of a multi-programmed workload.

We ran various multi-programmed workloads consisting of four applications among: *blackscholes*, *bodytrack*, *canneal*, *dedup*, *ferret*, *fluidanimate*, *raytrace*, *streamcluster*, *swaptions*, and *x264* and balanced their run times by re-execution.

Fig. 5 displays the efficiency curves of *ThermOS*, *Dimetrodon*, and DVFS. Performance (i.e., the ratio between the execution time without and with the intervention of DTM techniques) decreases linearly from 100 % to 75 % when setting the P-states through DVFS; however, temperature decreases are less predictable.

Dimetrodon employs a probabilistic, feedforward controller to drive ICI; the probabilistic nature of the controller and the absence of feedback make the behavior mostly unpredictable. Fig. 5 highlights the Pareto-optimal executions of *Dimetrodon*, which are slightly worse than those of *ThermOS*; however, the interpolation of all the executions is worse than that of DVFS. Conversely to *Dimetrodon*, *ThermOS* employs a formal feedback controller backed by a robust thermal model to drive ICI. The conjunction of these elements make the behavior of *ThermOS* highly predictable. Fig. 4b displays a performance decrease of $\approx 8\,\%$ with respect to Fig. 4a; this is expected considering Fig. 5 and a temperature decrease of $\approx 20\,\%$.

When cores operate at decreased clock frequency the relative latency of the memory hierarchy tends to decrease alongside with the bandwidth [19]. While the latter effect is negative and compromises the performance of memory-intensive applications, the former is positive for CPU-intensive applications since it lessens the effects of memory stalls. We believe P-states close to the highest do not allow DVFS to balance the number of instructions issued, which is known to have a higher correlation with temperature than the number of instructions retired [20], with an adequate decrease of power consumption.

It is well accepted that the dynamic power consumption of an integrated circuit made up of an ensemble of transistors scales as reported in Eq. (13); where a is some proportionality constant, C is the capacitance of a single transistor, V_{dd} is the supply voltage, f is the clock frequency, and n_t is the number of transistors that switches concurrently on average.

$$P_d \propto a \cdot C \cdot V_{dd}^2 \cdot f \cdot n_t \tag{13}$$

Until the beginning of 2000s, the supply voltage V_{dd} has decreased constantly with the ever-decreasing feature size; however, the shift from the micrometer to the nanometer realm prevents this from happening with the same pace as before. The supply voltage V_{dd} is bound to be twice as much as the threshold voltage V_{th}, which is not scaling down [21].

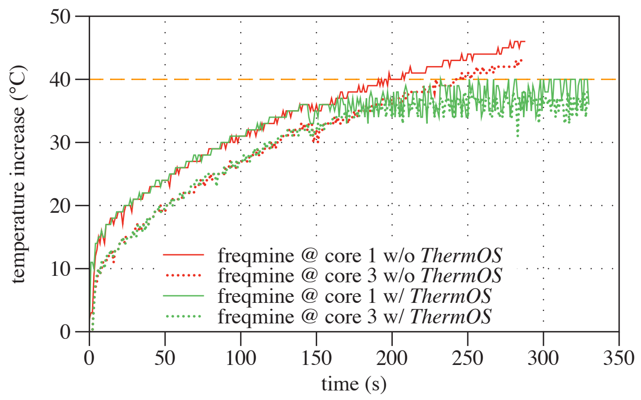


Fig. 6. Execution of a multi-threaded application on a commodity CMP. *ThermOS* constraints temperature below the threshold but introduces artificial critical paths making performance unpredictable.

DVFS is doomed to progressively lose its effectiveness since the clock frequency f will be the only difference among the available P-states and its weight is not comparable to that of the squared supply voltage V_{dd}^2. *ThermOS* already achieves better efficiency than DVFS when tackling the trade-off between performance and temperature for decrease of the latter up to $35\,\%$ and this margin is likely to increase in the foreseeable future. In fact, future efficiency curves for DVFS will most likely fall in the area below the current one (see Fig. 5).

D. Limitations

In this work we focus on the trade-off between performance and temperature for multi-programmed workloads consisting of single-threaded applications. Let us consider a different setting in which a CMP runs a multi-threaded application. As already pointed out, cores on a CMP heat up following different thermal profiles, thus requiring a more or a less aggressive ICI. Whenever this happens, one or more threads may be slowed down more than the others, thus putting in place an artificial critical path impairing synchronization and stretching the run time unpredictably. Fig. 6 displays this issue by means of two consecutive runs of *freqmine* executing with four threads. In this “unsupported” setting, *ThermOS* achieves a temperature decrease of $\approx 11\,\%$ at the cost of a performance decrease of $\approx 15\,\%$.

Being aware of this issue we plan on improving *ThermOS* so as to cope with synchronizations. A first naïve approach for finely-synchronized (e.g., barrier-based) multi-threaded applications requires ICI to work with the same timing and intensity among the cores running an application, doing so by choosing the idle time required by the core with the worst thermal profile. This is clearly sub-optimal from a performance standpoint. A more elaborated solution requires an augmented thermal model accounting for thermal interactions among cores since synchronizing ICI will greatly enhance its efficiency [13, 20]. An approach for coarsely-synchronized (e.g., lock-based) multi-threaded applications exploits previous work on scheduling for symmetric multi-processors and avoids throttling a thread inside a critical section.

VI. Related Work

Researchers proposed a variety of techniques to deal with temperature issues. We classify these techniques in three categories: architectural, micro-architectural, and software.

A. Architectural Approaches

Clock and power gating limit the distribution of the clock signal and the supply voltage, respectively. They decreases the dynamic and static power consumption, respectively, since the former prevents transistors from switching while the latter cut them off from the supply voltage. C-states exploit clock and power gating to decrease energy consumption.

Near/sub-threshold voltage (NTV/STV) designs [22] dramatically increase energy efficiency at the cost of severe drops of clock frequencies and single-threaded performance. With the shift from single to multi-core processors we assisted to the proliferation of multi-threaded applications. The adoption of NTV/STV designs moves the limits even further requiring embarrassingly multi-threaded applications, which are far from common. Hence, researchers disagree about the applicability and success of NTV/STV designs [21].

A recent architectural approach employs *c-cores* [23] to decrease energy consumption and power density by means of pre-synthesized application-specific co-processors. *c-cores* are promising but require a substantial paradigm shift at the computer architecture level since the most likely implementation require the adoption of reconfigurable fabrics.

B. Micro-Architectural Approaches

Micro-architectural approaches like instruction window sizing, issue width sizing, and instruction fetch toggling aim at limiting energy consumption by decreasing the number of instructions issued per clock cycle. Performance penalties due to the adoption of micro-architectural approaches can be amortized by orthogonal techniques like activity migration [24], which however requires additional transistors.

Brooks and Martonosi [14] propose a set of heuristics to drive instruction fetch toggling. Skadron et al. [16] show the applicability of the compact thermal model and formal feedback control to drive instruction fetch toggling achieving predictable behavior and the desirable properties control theory can guarantee. Jayaseelan and Mitra [25] harnesses instruction window and issue width sizing alongside with instruction fetch toggling and a neural network predictor to implement DTM.

Collectively, micro-architectural approaches can guarantee safety; however, being implemented at the lowest level of the hardware/software execution stack, they lack visibility and may impair the performance of critical pieces of software such as real-time and kernel-mode tasks, and interrupt request routines. In addition, most of these approaches are not available in commodity designs that need alternative software approaches.

C. Software Approaches

Common software approaches regard thermal-aware scheduling and DTM. Thermal-aware scheduling for large-scale computing systems involves migrating tasks from hot to cold islands [26], while thermal-aware scheduling for servers is concerned with tasks placing [20] and ordering [15].

Powell et al. [20] propose *Heat-and-Run*, a technique to perform assignment and migration of tasks to balance temperature across a CMP. *Heat-and-Run* is a thermal-aware scheduler that exploits simultaneous multi-threading (SMT) to co-schedule “complementary” tasks (e.g., one ALU-intensive and one FPU-intensive) on the same core and the availability of many cores to alternate heating and cooling phases.

Zhou et al. [15] propose *ThreshHot* and observes that tasks ordering actually matters. *ThreshHot* is a thermal-aware scheduler that orders tasks from “hot” (mostly CPU-intensive) to “cold” (mostly I/O-intensive) and schedules them from the hottest to the coldest. This schedule is guaranteed to minimize temperature at the end of an epoch. The goal of thermal-aware scheduler is minimizing temperature without degrading vital measures such as throughput and latency.

DTM, and hence *ThermOS*, is orthogonal to thermal-aware scheduling since the former tackles those settings in which the latter cannot prevent temperature from exceeding the threshold. Kumar et al. [27] propose *HybDTM*, which still exploits the “hot” and “cold” tasks classification but without following a “hot-to-cold” schedule. Whenever temperature exceeds the threshold, *HybDTM* throttles “hot” tasks first by lowering their priority, thus allowing “cold” tasks to use more processor time. *HybDTM* is meant for single-core processors and many of its considerations do not apply in the CMP realm.

kidled [12] is Google’s ICI implementation. It allows administrators to set a core-wide idle time over a time period. If the end of an interval draws near and the core has not been naturally idle for the requisite time, *kidled* injects idle time. *PowerClamp* [13] is Intel’s ICI implementation. Bailis et al. [4] propose *Dimetrodon*, a framework implemented inside the FreeBSD kernel that relies on probabilistic feedforward control and ICI as a means for decreasing temperature. Conversely to *ThermOS*, *kidled*, *PowerClamp*, and *Dimetrodon* are eager when injecting idle time. In addition, *ThermOS* leverages a thermal model and formal feedback control to drive ICI.

VII. Conclusions and Future Work

ThermOS proved effective in managing temperature during the execution of multi-programmed workloads and achieved better efficiency than both commodity and cutting edge DTM techniques. The evaluation shows that *ThermOS* can selectively affect applications within batch-style, multi-programmed workloads running on a commodity CMP. *ThermOS* also displays higher flexibility and better efficiency than DVFS for temperature reduction of up to 30 %. Moreover, the use of formal feedback control provides *ThermOS* with better predictability than *Dimetrodon*.

As future work we intend to both address the limitations highlighted in this paper (e.g., exploit technologies like SMT and HTT, execute multi-threaded applications, etc.) and integrate DTM with performance management [28–30] to guarantee service-level objectives.

Acknowledgment

The authors would like to thank Davide B. Bartolini, Gianluca C. Durelli, Alessandro A. Nacci, and the anonymous referees for their help and invaluable comments. This work was partially supported by the Swedish Research Council through the LCCC Linnaeus Center.

References

[1] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, “Temperature-Aware Microarchitecture: Modeling and Implementation,” in *ACM Trans. Archit. Code Optim.*, vol. 1, no. 1, 2004.

[2] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder, “Temperature Management in Data Centers: Why Some (Might) Like It Hot,” in *SIGMETRICS*, 2012.

[3] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, “The Case for Lifetime Reliability-Aware Microprocessors,” in *ISCA*, 2004.

[4] P. Bailis, V. J. Reddi, S. Gandhi, D. Brooks, and M. I. Seltzer, “Dimetrodon: Processor-Level Preventive Thermal Management via Idle Cycle Injection,” in *DAC*, 2011.

[5] M. Garrett, “Powering Down,” in *Commun. ACM*, vol. 51, no. 9, 2008.

[6] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd., Toshiba Corp., “Advanced Configuration and Power Interface Specification, Revision 5.0,” http://www.acpi.info/spec50.htm, 2011.

[7] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, “An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget,” in *MICRO*, 2006.

[8] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, “System Level Analysis of Fast, Per-Core DVFS using On-Chip Switching Regulators,” in *HPCA*, 2008.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in *PACT*, 2008.

[10] V. Pallipadi, S. Li, and A. Belay, “cpuidle – Do nothing, efficiently. . . ,” in *Linux Symposium*, 2007.

[11] Intel Corp., “Intel Xeon Processor 3500 Series: Datasheet, Volume 1,” http://www.intel.com/content/www/us/en/processors/xeon/xeon-3500-series-vol1-datasheet.html, 2009.

[12] J. Corbet, “Idle cycle injection,” http://lwn.net/Articles/383368/, 2010.

[13] ——, “Intel PowerClamp Driver,” https://lwn.net/Articles/528124/, 2012.

[14] D. Brooks and M. Martonosi, “Dynamic Thermal Management for High-Performance Microprocessors,” in *HPCA*, 2001.

[15] X. Zhou, J. Yang, M. Chrobak, and Y. Zhang, “Performance-Aware Thermal Management via Task Scheduling,” in *ACM Trans. Archit. Code Optim.*, vol. 7, no. 1, 2010.

[16] K. Skadron, T. Abdelzaher, and M. R. Stan, “Control-Theoretic Techniques and Thermal-RC Modeling for Accurate and Localized Dynamic Thermal Management,” in *HPCA*, 2002.

[17] W. S. Levine, *The Control Handbook*. CRC-Press, 1996.

[18] C. Yang and A. Orailoglu, “Processor Reliability Enhancement through Compiler-Directed Register File Peak Temperature Reduction,” in *DSN*, 2009.

[19] R. Schöne, D. Hackenberg, and D. Molka, “Memory Performance at Reduced CPU Clock Speeds: An Analysis of Current x86-64 Processors,” in *HotPower*, 2012.

[20] M. D. Powell, M. Gomaa, and T. N. Vijaykumar, “Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through the Operating System,” in *ASPLOS*, 2004.

[21] L. Chang and W. Haensch, “Near-Threshold Operation for Power-Efficient Computing? It Depends...” in *DAC*, 2012.

[22] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, “Near-Threshold Voltage (NTV) Design - Opportunities and Challenges,” in *DAC*, 2012.

[23] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, “Conservation Cores: Reducing the Energy of Mature Computations,” in *ASPLOS*, 2010.

[24] S. Heo, K. Barr, and K. Asanovic, “Reducing Power Density through Activity Migration,” in *ISLPED*, 2003.

[25] R. Jayaseelan and T. Mitra, “Dynamic Thermal Management via Architectural Adaptation,” in *DAC*, 2009.

[26] J. D. Moore, J. S. Chase, P. Ranganathan, and R. K. Sharma, “Making Scheduling “Cool”: Temperature-Aware Workload Placement in Data Centers,” in *ATC*, 2005.

[27] A. Kumar, L. Shang, L.-S. Peh, and N. K. Jha, “HybDTM: A Coordinated Hardware-Software Approach for Dynamic Thermal Management,” in *DAC*, 2006.

[28] F. Sironi, D. B. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, and M. D. Santambrogio, “Metronome: Operating System Level Performance Management via Self-adaptive Computing,” in *DAC*, 2012.

[29] D. B. Bartolini, F. Sironi, M. Maggio, R. Cattaneo, D. Sciuto, and M. D. Santambrogio, “A Framework for Thermal and Performance Management,” in *MAD*, 2012.

[30] D. B. Bartolini, R. Cattaneo, G. C. Durelli, M. Maggio, M. D. Santambrogio, and F. Sironi, “The Autonomic Operating System Research Project: Achievements and Future Directions,” in *DAC*, 2013.