Thermal-Aware Scheduling and Load Balancing Using Predictive Energy Consumption Models in High-Performance Multicore Systems

Adam Lewis and Nian-Feng Tzeng

Center for Advanced Computer Studies, University of Louisiana at Lafayette, Louisiana 70504
{awlewis, tzeng}@cacs.louisiana.edu

Abstract*—Modern processors crudely manage thermal emergencies through Dynamic Thermal Management (DTM), where the processor monitors the die temperature and dynamically adjusts the processor voltage and frequency (DVFS) to throttle down the processor when necessary. However, DVFS tends to yield marked degradation in both application performance and system reliability. Thus, pro-active scheduling techniques that avoid thermal emergencies are preferable over reactive hardware techniques like DTM. Based on our previously introduced thermal Chaotic Attractor Predictors (CAPs, which take into account key thermal indicators and system performance metrics for overall energy consumption estimation within a given power and thermal envelope), we have developed and evaluated an effective thread scheduler for multicore systems. Besides CAPs, our scheduler makes use of two basic principles to minimize server energy consumption: (1) selecting the thread with the least probability of causing a DTM in the subsequent time quantum, for execution on the next available core, and (2) migrating execution threads on thermally overextended cores to other cool cores via load balancing so as to observe the thermal envelope. Our developed scheduler is evaluated in practice to assess its potential advantage resulting from thermal-awareness by incorporating CAPs (for temperature prediction) and basic principles (for energy reduction) into the existing scheduler in the FreeBSD operating system. Our implemented scheduler is run on a server with the Intel Xeon processor for gathering measures of interest (including die temperature readings and execution times) when benchmark codes from PARSEC and SPEC CPU2006 suites are executed. The gathered results reveal that our proposed scheduler exhibits reduction in mean core die temperatures by up to 12°C under PARSEC benchmarks (which have multithreaded workloads) and by up to 3.3°C under mixes of SPEC CPU2006 benchmarks for concurrent execution on all four server cores, while experiencing only 1% to 4% performance degradation.

I. Introduction

Modern processors crudely manage thermal emergencies through Dynamic Thermal Management (DTM), where the processor monitors the die temperature and dynamically adjusts the processor voltage and frequency (known as Dynamic Voltage and Frequency Scaling (DVFS)) to throttle down the processor whenever necessary. However, the use of DVFS tends to cause significant negative impacts on application performance and system reliability [?], [?], [?].

This paper introduces effective scheduling for preventive thermal management that minimizes server energy consumption by (1) selecting the subsequent thread with the smallest thermal impact in the next time quantum to execute on an available core, and (2) relocating threads run on thermally overextended cores to other available cores for load balancing. As opposed to prior pursuit of thermal-aware scheduling [?], [?], [?], [?] that aimed to bound temperatures below critical thresholds, our work considers how to schedule high workload for die temperature management across all cores.

The current generation of operating systems treats those cores in multicore and virtualized multi-threaded processors (like those with Intel's HyperThreading technology) as distinct logical units, called *logical cores*, which are scheduled independently. However, dependency and contention exist in shared resources among those logical cores, and hence they need to be taken into account upon their scheduling to address performance and energy efficiency. One approach based on software optimization at the application level aimed to make codes themselves more aware of existing dependencies [?]. While effective, such an approach may not be applicable in all cases and does not possess economies of scale. It thus calls for the need of intelligent, thermal-aware load balancing and scheduling within the operating system, achievable via modeling full-system energy consumption based on computational load for effectively predicting future energy consumption and its associated thermal change.

To this end, we arrive at a thermal model which relates server energy consumption to the overall thermal envelope, establishing an energy relationship between workload and overall system thermodynamics. According to our analysis of experimental measurements of key processor performance counter readings and performance metrics, it is found that the measured readings and metrics do not possess linearity and are *chaotic in nature*. Hence, our thermal model, based on a thermal Chaotic Attractor Predictor (tCAP), takes into account key thermal indicators (like ambient temperatures and die temperatures) and system performance metrics (like performance counters) for system energy consumption estimation within a given power and thermal envelope. This work demonstrates that effective scheduling can result from taking advantage of our devised tCAP when dispatching jobs to confine server power consumption within a given power budget and thermal envelope while avoiding detrimental impact on thread execution performance.

Dealing with all cores in a multicore system, a thermal-aware thread scheduler is highly desirable to ensure that all of the cores in such a system are kept equally busy, realized by migrating threads over all the cores (no matter whether in the same processors or different ones) when necessary for load balancing in the system. Existing schedulers aid in processor thermal management by dispatching workloads to cores as tightly as possible within the same processors to expose more opportunities for power management via shutting down unused cores. However, computation-bound server workload fully utilizes available system cores in most times, thereby rendering such schedulers ineffectual. Our Thermal-Aware Scheduler (TAS) addresses this problem by thermally balancing system workload with as little performance degradation as possible.

Our TAS is demonstrated and evaluated by adding thermal-awareness to the existing scheduler in the FreeBSD operating system executed on Intel Xeon (Woodcrest) processors. Benchmarks from SPEC CPU2006 and PARSEC suites which represent typical application server workloads, are chosen to evaluate our TAS. The gathered results unveil that TAS achieves reduction in mean core on-die temperatures by up to 12°C under PARSEC benchmarks (which have multi-threaded workloads) and by up to 3.3°C under mixes of SPEC CPU2006 benchmarks for concurrent execution on all four server cores, while experiencing only 1% to 4% performance degradation. Our TAS compares favorably with a recent energy-aware scheduling technique, which gets core temperature reduction by 4°C (from 63°C down to 59°C) when the parallel benchmark of Charm++ was executed on a physical 4-core Intel Xeon 5520 processor (similar to our testbed processor) [**?**].

After an overview on pertinent background and related work is provided in Section **??**, this article introduces our proposed thermal model in Section **??** and explains in Section **??** how the model can be used for predictive purposes. Section **??** details the design of our TAS, which is experimentally evaluated in Section **??**, with obtained evaluation results included and discussed as well. Section **??** offers our conclusion.

II. Background and Related Work

The thread scheduler of an OS kernel is responsible for placing threads in the dispatch queue, deciding which threads to run next on available logical cores, and managing thread migration to balance the workload amongst all cores. In addition, a scheduler must fulfill two major applications requirements: (1) making equal progress on all threads of a given application, and (2) exploiting as much hardware parallelism as possible [**?**]. Traditional server load balancing has various assumptions about workload behavior when making thread placement decisions. In particular, interactive workloads are assumed to involve independent tasks that remain quiet for extended periods. Server workloads, on the other hand, are assumed to contain large numbers of threads which are highly independent of each other and use synchronization objects to ensure mutual exclusion on small data items. Modern operating systems (such as Linux and Solaris) make load balancing more power-aware by placing workloads to cores as tightly as possible (inside fewest processors possible), thereby presenting more opportunities for power management software to shutdown unused resources [**?**], [**?**], [**?**].

The following subsections provide in sequence, background and prior work pertinent to thermal modeling, scheduling with load balancing support, and chaotic behavior of energy consumption observed in our prior article.

A. Thermal Modeling

Existing thermal models all required to fit targeted mathematical expressions based on time-series observations of temperatures during the course of executing various workloads. Expression coefficients were estimated by minimizing total least square errors between modeling results and actual temperature readings. After proper calibration, such a mathematical expression becomes a model for extrapolating future temperatures. Different modeling techniques have been devised. In particular, a model based on integer linear programming was adopted by an earlier task scheduler [**?**], which aimed to meet real-time deadlines while minimizing hot spots and spatial temperature differentials across the die. Meanwhile, dynamic thermal modeling includes Heat-and-Run [**?**], HybDTM [**?**] and ThreshHot [**?**],[**?**]. Heat-and-Run distributes work among available cores until the DTM events arise, and it then migrates threads from the overheated cores to other cool cores. On the other hand, HybDTM enhances DTM with a thread migration strategy which lowers the priority of jobs executed on hot cores. Separately, ThreshHot employs an on-line temperature estimator to determine the proper order to schedule threads across cores, favoring those threads which cause the greatest temperature hikes while avoiding DTM events to occur. Schedulers based on prior thermal modeling all rely on readings of hardware performance counters and temperature sensors. They can be improved by analyzing on-die thermal variations to aid in system power and thermal management [**?**], [**?**], [**?**].

However, preceding techniques are reactive to the temperature approaching the DTM threshold rather than trying to avoid reaching that temperature in the first place. A proactive solution with multi-tier prediction was suggested earlier [**?**], where a core level predictor was employed to convert temperature observations to operating frequency estimates while a control-theoretic based scheduler was followed at the socket level for process level scheduling. Separately, a scheduling policy for sorting the tasks in each core's run queue according to memory intensity was considered so as to schedule memory-bound tasks at slower frequencies [**?**], [**?**]. Later, the process scheduling policy was modified in [**?**] to allocate time slices following (1) the contribution of each task to the system power consumption and (2) the current processor temperature. A similar approach made use of idle cycle injection to manage CPU time slice allocation, aiming to maintain a lower average temperature over time, as opposed to managing temperatures against a critical threshold. Meanwhile, Cool Loop [**?**] and Dimentrodon [**?**] address a lack of heat slack by inserting additional cycles into the task scheduling to create thermal slack, naturally leading to performance degradation. A variation of preceding schemes relied on system level compiler support to insert a run-time profiling code into applications for providing hints on the thermal intensity of a task [**?**]. However, such an approach works ineffectively under many server cases where the slack in deadlines usually is unavailable.

B. Earlier Scheduling with Load Balancing Support

Modern multiprocessor operating systems (such as Windows, Linux, Solaris, and FreeBSD) often take a two-level approach to task scheduling for maximized system resource utilization. Such an approach uses a distributed run queue and follows fair scheduling policies to manage each core at the first level. Its second level balances the work load by redistributing tasks across the queues. In particular, the FreeBSD ULE scheduler [?], [?], [?] uses a combination of push and pull thread migration for load balancing. *Push migration* scans the run queues associated with each processor every 500 milliseconds to pick the most-loaded and least-loaded logical cores before equalizing their run-queues. The processor then attempts to pick a core in the same processor group so as to minimize the migration cost. In *pull migration*, a processor checks if it has excess work in its run queue and also if another processor in the system is idle, before adding new thread to its run queue. The existence of an idle processor triggers an inter-processor interrupt to migrate the new thread to the idle processor. Such two-level schedulers work under three assumptions: (1) threads are independent, (2) load is governed by queue length, and (3) locality exists and is important [?]. In practice, however, common servers often have the following characteristics: (1) their threads are logically related, with data and control dependencies among threads, and (2) their threads have equally long life spans [?], rendering previous two-level scheduling ineffective.

Meanwhile, existing operating systems have made their load balancing schemes more power-aware by taking advantages of their power management drivers which intend to find as compact an allocation of processes to run-queues as possible [?], [?], [?], [?]. This way presents more opportunities for the power management software to shutdown unused resources. Such power-aware load balancing, while effective for interactive workloads, works only if system resources are not completely utilized; it becomes nonviable if the system workload is high (common to high-performance servers).

C. Chaotic Behavior of Energy Consumption

An analytical model of server energy consumption was built earlier [?], [?] by modeling energy consumption as a function of the work done by the system in executing its computational tasks and of residual thermal energy given off by the system in doing that work. The resulting dynamic system expresses energy consumption in the time domain as follows:

$$E_{system} = f(E_{proc}, E_{mem}, E_{em}, E_{board}, E_{hdd}) \qquad (1)$$

where each of the terms in the above equation is defined as: (1) E_{proc}: energy consumed in the processor due to computations, (2) E_{mem}: energy consumed in the DDR SDRAM chips, (3) E_{em}: energy taken by the electromechanical components in the system, (4) E_{board}: energy consumed by peripherals that support the operation of the board, and (5) E_{hdd}: energy consumed by the hard disk drive during the system's operation.

The continuous system in Eq. (**??**) can be viewed as a multi-variate differential equation in the time domain that can be estimated using a time series representation. This time series is constructed by considering (1) an initial energy state E_{system} at time $t = 0$ and (2) a set of physical predictors that approximate the values of E_{proc}, E_{mem}, E_{board}, and E_{hdd} at the next interval $t + \Delta t$.

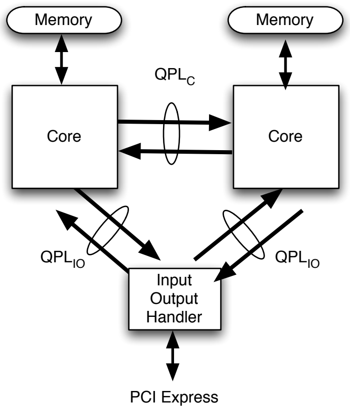

Fig. 1: Intel Xeon (Woodcrest) architecture.

TABLE I: PeCs and performance metrics for Intel Xeon server

Variable	**Measurement**
Application length	
IR	Instructions Retired
Application data set	
QPL_C	Transactions on the QPLs between Cores
QPL_{IO}	Transactions on QPLs for IO Handler
CM_0	Last-level Cache Misses due to Core 0
CM_1	Last-level Cache Misses due to Core 1
CM_2	Last-level Cache Misses due to Core 2
CM_3	Last-level Cache Misses due to Core 3
D_r	Disk bytes read
D_w	Disk bytes written
Physical core temperature	
T_{C_0}	Core 0 Die Temp
T_{C_1}	Core 1 Die Temp
T_{C_2}	Core 2 Die Temp
T_{C_3}	Core 3 Die Temp
System temperature	
T_{A_0}	Ambient Temp 0
T_{A_1}	Ambient Temp 1
T_{A_2}	Ambient Temp 2
Electromechanism	
F_C	Memory Cooling Fan Speed
F_{M2a}	Memory Cooling Fan Speed 2a
F_{M2b}	Memory Cooling Fan Speed 2a
F_{M3a}	Memory Cooling Fan Speed 3a
F_{M3b}	Memory Cooling Fan Speed 3b
F_{M4a}	Memory Cooling Fan Speed 4a
F_{M4b}	Memory Cooling Fan Speed 4b
F_{M5a}	Memory Cooling Fan Speed 5a
F_{M5b}	Memory Cooling Fan Speed 5b

Observations from each time series are measured using (1) the appropriate PeCs for the targeted processor (2) operating system kernel virtual memory statistics, and (3) processor die temperatures and chassis ambient temperatures. Twentyfour server measures are input to our predictor for energy consumption, as listed in Table **??**. They are classified into five groups, each associated with one server energy contributor. As one contributor, application length is estimated in each time period by the quantity IR, the total number of retired instructions of the thread. Among the contributor of "application data set" listed in Table I, QPL_C and QPL_{IO} are relevant to QuickPath Links (depicted in Fig. **??**), and they are associated with E_{proc} and E_{mem}, respectively. In practice, however, there is just one single PeC for holding aggregated QPL_C and QPL_{IO} together. The next four CM_i measures indicate the total L3 (last-level) cache miss counts due to Core i, i = 0, 1, 2, or 3, and they determine E_{mem}. As the contributor of "physical core temperature," the core die temperature measures are pertinent to E_{proc}. The subsequent three measures dictate E_{board}, obtained from 3 temperature sensors placed on the board for ambient temperature readings. Finally, the last nine measures contribute to "electromechanism" and determine E_{em}, offering speed information of those nine memory cooling fans, to constitute MP_{em}. As a result, each observation for the Intel server at time t comprises the 24 measures as listed in the table.

D. Limitations of Existing Scheduling

Current power management software that utilizes DVFS techniques to address DTM events has been effective in addressing thermal emergencies [**?**], [**?**], commonly implemented in modern server processors [**?**], [**?**]. However, handling DTM through DVFS can be problematic due to issues with program phase behavior and contention for shared resources [**?**], [**?**], resulting directly from slow transitions between the active and the idle device states and also from inability to access resources associated with idle processors. When the power phase changes frequently, abundant thermal variations among cores within the processor occurs, leading to decreased reliability [**?**], [**?**], [**?**]. For example, the Intel XScale processor was reported to decrease its component MTTF (Mean Time to Failure) by 12% to 34%, depending upon the selected power management strategy [**?**].

Work migration for energy savings and thermal management has a long history in the SMP, SMT, and CMP environments [**?**], [**?**], [**?**], [**?**]. A study of OS-level thermal migration using Linux on the IBM POWER5 processor [**?**] discovered that the rise and fall times of core temperatures vary in the order of hundreds of milliseconds. As most operating systems choose scheduler ticks to be of 10 ms or less, it often is impossible to react to thermal conditions before a critical state is reached. As a result, three improvement mechanisms for managing thermal states have been pursued: (1) core hopping for leveraging spatial heat slack, (2) task scheduling for leveraging temporal heat slack, and (3) SMT scheduling for leveraging temporal heat slack. In the presence of slack, each of those mechanisms may reduce core die temperatures by 3 to 5° C on an average, at the expense of 3% mean performance degradation [**?**], [**?**]. However, in the absence of slack commonly found under heavy workloads in high-performance servers, the three mechanisms becomes ineffective, calling for suitable scheduling with thermal awareness proactively.

III. System Thermal Model

In this section, we introduce a system thermal model built upon what was reviewed in Section **??** to address the thermal domain by first considering how to extend Eq. (**??**) to account for energy consumed during application execution.

An application is composed of p execution threads, each associated with a data set sized d_i, for $1 \leq i \leq p$, in a processor. The total data associated with A is the sum of the data associated with its component threads:

$$D_A = \sum_{i=1}^{p} d_i. \tag{2}$$

We assume that the activities take place (1) in a staging area, which contains both main and virtual memory operating space, and (2) in the processor with its cores and their associated caches. The execution time measurement includes computation time and the time to move application data from the staging area (on peripherals off the chip like DRAM and HDD) to a computation or operation area (on the chip, such as the caches and the cores).

Each application A with the problem size of D_A involves workload $W(\tau_i, d_i, t_i)$, for $1 \leq i \leq p$, comprising two components: (1) a count of the operations performed by the computational core, and (2) the count of communication operations required for transferring data, instructions, and data coherency and book-keeping functionality traffic. They are measured in terms of the number of bytes operated upon or transferred over, during the course of completing those instructions retired by the logical core for each thread p. Thus, energy consumed by executing application A with data set D_A can be expressed as:

$$E_A(A, D_A, t) = \sum_{i=1}^{p} W(\tau_i, d_i, t_i) \tag{3}$$

for $1 \leq i \leq p$, where τ_i is an execution thread involved in application A, d_i is the corresponding data set for that thread, and t_i is its thread execution time.

In order to relate system energy expenditure (upon application execution) to corresponding joule heating, we define the term of "Thermal Equivalent of Application" (TEA), as the electrical work converted to heat in running the application, leading to die temperature change and ambient temperature change of the system. Hence, the TEA of application A is expressed by:

$$\Theta(A, D_A, T, t) = \frac{E_A(A, D_A, t)}{\lim\limits_{T \to T_{th}} J_e(D_A, \Psi_{cp})(T - T_{nominal})}, \tag{4}$$

where T_{th} denotes the threshold temperature at which a thermal emergency event will occur, and $T_{nominal}$ refers to the nominal temperature as reported by the system when it is in a quiescent state, i.e., only the operating system is running without any application execution. The term J_e is the "electrical equivalent of heat" for the chip, which reflects the *informational entropy* of the system due to processing the data bits during application execution and to the black body thermal properties of the chip packaging as well as the cooling mechanisms around the chip, as defined by the parameter Ψ_{cp}. The value of Ψ_{cp} is an ambient thermal characterization parameter provided by the hardware manufacturer to relate temperature to power for cooling purposes [**?**]. Thus, TEA is a dimensionless quantity, with both denominator and numerator expressing work done or energy consumed in finishing an application.

We combine these metrics into achieved performance per unit energy consumed by the chip:

$$C_\theta(A, D_A, T, t) = \frac{\Theta(A, D_A, t)}{E_A(A, D_A, t)} \quad (5)$$

This normalized quantity indicates the "cost" of executing an application on a given processor, with $E_A(A, D_A, t)$ obtained from energy consumption of individual physical components (processor, DRAM units, HDD, motherboard, and electrical/-electromechanism) given by Eq. (**??**).

A. *Thermal Chaotic Attractor Predictors*

We use the measured PeCs listed in Table **??** to estimate the quantities of $\Theta(A, D_A, t)$ and $E_A(A, D_A, t)$ in Eq. (**??**). Specifically, thread length is estimated in each time period by the PeC measure of IR, namely, the total number of instructions retired during that thread execution. The total data amount associated with thread execution is measured by summing up (1) the data bytes moved across the QuickPath links on the processor (reflected by QPL_C and QPL_{IO}), (2) last-level cache misses ($CM_i, 1 \leq i \leq 4$), and (3) data read from or written to disks. as listed under the contributor of "application data set" in Table **??**. System temperature is measured using the ambient temperatures reported by the system, as listed under the contributor of "system temperature."

Following the procedure reviewed in Section **??** (and detailed in [**?**]), we intend to create a thermal Chaotic Attractor Predictor (tCAP) for Eq. (**??**) as an effective means for predicting the thermal behavior during application execution. To this end, an analysis on the measurement data collected from our test systems is first performed to confirm that the behavior of the time series data can be attributed to a certain form of chaotic behavior. It involves evaluating system (i.e., testbed) sensitivity to initial conditions, by calculating the Lyapunov exponents of the time series data observed while running various benchmarks (stated in Section V.B) on the testbed. The Lyapunov exponent quantifies the sensitivity of a system such that a positive Lyapunov exponent indicates that the system is chaotic [**?**]. We have found a positive Lyapunov exponent when performing this calculation on our data set, ranging from 0.019 to 0.051 on our Intel test server, as listed in Table **??**. Therefore, our collected data has met the first and the most significant criterion to qualify as a chaotic process.

The second indication of the chaotic behavior for time series approximations expressed by Eq. (**??**) is an estimate of the Hurst parameter H for the time series observations. If the value of the Hurst parameter is greater than 0.5, an increment in the random process is positively correlated and long range dependence exists in the case of time series [**?**]. In a chaotic system, a value of H approaching 1.0 indicates the presence of self-similarity in the system. As demonstrated in Table **??**, the time series data collected in our experiments all have H values close to 1.0, ranging from 0.95 to 0.99 for the Intel server in our test environment.

TABLE II: Chaotic behavior in core die temperature time series

	Hurst Exp. (H)	Lyapunov Exp. (λ)
Core 0	0.99	0.051
Core 1	0.98	0.019
Core 2	0.97	0.034
Core 3	0.95	0.040

IV. Proposed Thermal Aware Scheduler

The scheduler in an operating system is responsible for making two decisions in each time quantum: (1) thread scheduling, i.e., deciding the next thread to run on an available core and (2) load balancing, namely, distributing workload evenly across all cores, with existing implementations mostly focusing on performance. Our TAS (Thermal Aware Scheduler) incorporates a heuristic scheduling algorithm in a popular scheduler (i.e., ULE in the FreeBSD operating system) for thermal stress reduction on a multicore processor while meeting the SPMD requirements of equal execution progress and maximum parallelism exploitation.

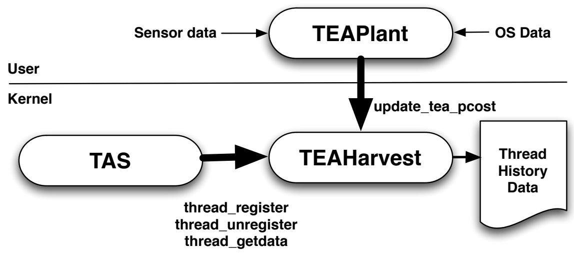

Fig. 2: TEAPlant and TEAHarvest data collection.

A. *Thermal Predictors*

We enhance the existing FreeBSD operating system to maintain information required by the thermal estimator. Our design is based on the concept of Task Activity Vectors (TAVs) introduced earlier [**?**], with a vector for each kernel thread to store the required history in order to make sound prediction. Generally, the more additional space is employed for history maintenance, the higher benefit our thermal scheduling gains.

The high-level design of TAS is shown in Fig. **??**. A user-level daemon process collects required information to compute the time-series predictions for Θ and C_θ. Temperature readings are collected by this process from the digital temperature sensor associated with a core. Similarly, processor performance counters are gathered by the same process, with both sets of metrics used to generate estimates. Estimates are posted via a system call interface to a device driver that collects the data for use by the currently executing thread. The scheduler queries this driver via a kernel function call interface when making scheduling decisions to determine the Θ and C_θ estimates associated with a thread.

B. *Thread Scheduling*

The scheduler uses the cost predictor for C_θ to predict a thread's impact on core temperature and adjust the thread priority as required to prevent an increase in core temperature. TAS maintains three queue structures per core: an idle queue, a current queue, and the next queue. The core executes all work on the current queue and then swaps its current queue and its next queue. For performance reasons, our implementation follows the convention used by the existing FreeBSD ULE scheduler, placing real-time and interrupt threads on the current queue. This is a reasonable limitation given that real-time thread deadlines must be satisfied and interrupt service routines are typically short in length.

All other threads are scheduled in terms of an interactivity score. As with the ULE scheduler, the interactivity of a process is determined by the formulas of

$$I = \frac{S}{\frac{SL}{RUN}} \tag{6}$$

and, for cases where thread's run time exceeds its sleep time,

$$I = \frac{S}{\frac{SL}{RUN}} + S \tag{7}$$

where I is the interactivity score, S, the scaling factor of the maximum interactivity score divided by two, and SL and RUN refer respectively to the cumulative sleep and run times for the thread. Threads whose scores fall below a predefined threshold are considered interactive and are assigned to the current queue, while all other threads are assigned to the next queue.

For our TAS, the interactivity score I is scaled by the predicted value of C_{θ}, normalized to a percentage value. It was shown previously [**?**] that the greatest thermal benefit occurred when a scheduler favored the thread which moved the core temperature as close as possible to the DTM threshold without actually triggering a DTM event. TAS achieves a similar effect by scaling the interactivity of a thread its normalized execution cost, thereby giving less "thermally costly" threads greater opportunities for execution so as to moderate the processor temperature. Penalizing more thermally costly threads reduces the opportunities for such threads to gain access to logical cores, presenting similar advantages of techniques which artificially inject slack into thread scheduling but without extra scheduling overhead incurred to those techniques.

C. Load Balancing

Load balancing distributes workload evenly across the available logical cores, with current implementation mostly aiming to maximize performance. This work applies load balancing to minimize thermal stress while seeking best performance. Specifically, TAS extends push migration by organizing system cores into "thermal clans" based upon the temperature and execution frequency. Specifically, a local core is assigned to one of the three thermal clans: Hot, Warm, and Cold, if its on-die temperature is respectively 90% or higher, between 75% and 90%, and below 75% of the DTM threshold temperature. Note that if a physical core supports $\alpha threads, \alpha logcalcoreswillresultfromthephysicalcore, withtheirtemperaturesallequaltothatofthephysicalcore.Inaddition, logic$ $levelcategorizationallowsTAStomanageworkloaddistributionbetterfromthethermalandperformanceperspectives, bymigratingw$

For performance reasons, information about thermal clans of all logical cores is maintained by the TEAHarvest thermal predictor driver. The driver allocates threads to local cores for execution, according to thermal efficiency and cost estimates. TAS scheduler queries the driver to determine whether a core will move towards the DTM threshold temperature if a thread becomes ready to execute on one of its logical core. In this way, TAS predicts whether a thread moves its assigned core closer to a DTM event and adjusts the core's run-queue accordingly to prevent DTM occurrence.

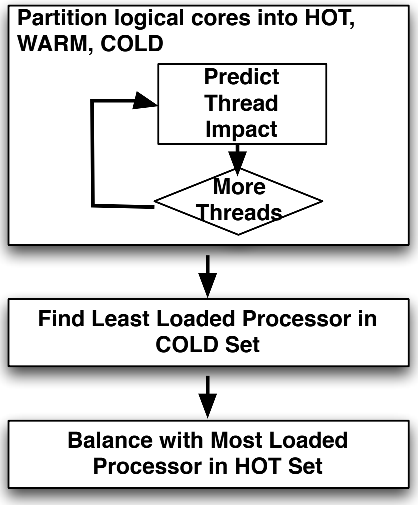
Partition logical cores into HOT, WARM, COLD
Predict Thread Impact
More Threads
Find Least Loaded Processor in COLD Set
Balance with Most Loaded Processor in HOT Set

Fig. 3: Design of thermal load balancing in TAS.

On a periodic basis (i.e., once in every 500 ms), the TAS scheduler executes the algorithm listed in Fig. **??** to balance the workload among logical cores, giving sufficient times for overtaxed resources to thermally recover. For each thread in the run queue, TAS uses the tCAP to estimate the value of Θ for each thread in the queue and predict the resulting change in temperature if this thread were to execute. It then moves the thread with the greatest temperature impact to the least loaded logical core in the "Cold" clan. This way results in workloads being moved away from thermally stressed logical cores, while maintaining execution performance.

Periodically, the system reads the temperatures of it all logical cores and, if required, also moves a logical cores to a different thermal clan. It should be noted that the time required for a logical core to recover from a thermal event is significantly longer than the interval used for thread scheduling [**?**]. This allows our TAS to use a much larger interval (2 seconds) between scans (across core temperatures) to determine if any logical core must be assigned to a new thermal clan for better scheduling outcomes.

V. Experimental Evaluation and Result Discussion

We have evaluated our TAS (Thermal Aware Scheduler) using the FreeBSD operating system run on a commodity server. Our TAS implementation modified the existing push migration in FreeBSD's ULE scheduler [**?**] to take into account both thermal behavior and system performance, as elaborated in Section **??**. PeC data were collected at the user-level through standard tools provided for the data collection purpose by

FreeBSD (i.e., the `coretemp` and `hwpmc` kernel extensions). They were collected and collated by a FreeBSD kernel extension, made available to the operating system scheduler for query when making scheduling decisions (Fig. **??**)

The processor thermal model was calibrated by measuring behavior outcomes of the testbed with TAS at idle and under high load stress using common utilities from the FreeBSD regression test suites and software collections. The CPU-related behavior of our scheduler was then characterized using integer and floating point benchmarks from the SPEC CPU2006 [**?**] benchmark suite. Benchmarks from the Princeton Application Repository for Shared-Memory Computers (PARSEC) suite [**?**] were then evaluated on the testbed to assess TAS in terms of key metrics of interest (i.e., die temperature and execution completion time) under high parallelism in the thread level.

Processor used in evaluation

	Dell Precision 490
CPU	Intel Xeon 5300 (Woodcrest)
CPU L2 cache	4MB
Memory	8GB,DDR2 667Mhz ECC
Internal disk	500GB
Network	1x1000Mbps
Video	NVIDA Quadro FX3400

A. Experiment Setup

Experimental evaluation was conducted on our testbed running TAS, with its hardware specified in Table **??**. The key metrics of interest were gathered during the course of application execution. Power consumed was measured by a WattsUP power meter [**?**], connected between the AC Main and the server testbed. The power meter measured the total and average wattage, voltage, and amperage over the run of a workload. The internal memory of the power meter was cleared at the start of each run and the measures collected during the runs were downloaded (after execution completion) from the meter's internal memory into a spreadsheet.

B. Benchmark Selection

Given processor components affect the thermal envelope in different ways [**?**], we selected our benchmarks carefully from SPEC CPU2006 and PARSEC suites (in addition to the FreeBSD stress testing codes) for TAS evaluation to satisfy three objectives: (1) sufficient coverage of the functional units in the testbed processor, (2) varying levels of CPU and memory intensity to reflect real-life scenarios, and (3) reasonable applicability to problem space.

Benchmarks chosen for our evaluation satisfy the first objective in two ways. The FreeBSD stress testing codes operate by repeating a tight set of integer and floating-point instructions while doing multiple reads and writes of large memory blocks. This way stresses the integer and floating-point units in the processor while engaging in large amounts of cache activities. Furthermore, the integer and floating-point benchmarks from the SPEC CPU2000 suite are chosen in different combinations for execution on the logical cores of the testbed simultaneously. This way of combining integer and floating-points benchmarks diversely for concurrent execution (as listed in Table V) stresses the functional units of the processor differently.

The second objective is met by selecting benchmarks from the SPEC CPU2006 and PARSEC suites that exhibit varying levels of CPU and memory intensity. It is demonstrated by executing (1) benchmarks singly to focus on each functional unit in the processor, and (2) various mixes of integer and floating-point benchmarks allocated across all logical cores in the system. Finally, the benchmarks were chosen from SPEC CPU2006 and PARSEC suites based on their degrees of fit into the problem space. Each benchmark represents an application typical for high-performance system execution.

C. tCAP Establishment

Since our TAS relies on tCAP (thermal Chaotic Attractor Predictor) of the testbed for effectively predicting its thermal behavior during application execution, it is required first to establish tCAP using suitable applications. In our earlier work, a few SPEC CPU2006 benchmarks were used as training applications for deciding the coefficients of the chaotic attractor predictor [**?**]. Here, we instead employed two applications which were expected to yield the extreme thermal results for training execution: the idle scenario and the most stress scenario. An idle application referred to no application thread ready for execution and thus the testbed runs only the OS threads, whereas the most stress application leads to the highest processor utilization coupled with heaviest memory activities, achieved by launching multiple instances of the FreeBSD `cpuburn` stress testing codes for concurrent execution, one per core with symmetric multiple threads disabled. The `cpuburn` stress test code implements a single-threaded infinite loop with a sequence of integer and floating-point operations and large blocks of memory reads and writes to thermally stress the testbed system, driving it close to a thermal emergency. The use of these two extreme thermal scenarios is preferred over employing SPEC CPU2006 benchmarks for tCAP establishment, since any typical application execution will lies between the two extremes, as far as the coefficients of tCAP are concerned.

These two training scenarios run for 600 seconds, with PeCs sampled at an interval of $t = 5$ seconds for establishing tCAP. The procedure for establishing tCAP is the same as what was described earlier [**?**], and it involved three steps. First, the training set is constructed by consolidating the observed PeCs into a time series using geometric means. In the second step, the Takens Embedding Theorem [**?**] is applied to find delay embedding, with the nearest neighbors algorithm then employed to identify the set of attractors using the embedded set. Finally, our tCAP is established by solving the linear least squares problem resulted from fitting a multi-variate polynomial to the attractor set. The time complexity of establishing tCAP equals O(n^2), where n is the number of sampled PeCs [**?**]. Note that the establishment of the attractor set for tCAP is required only once for the given testbed (i.e., any processor of interest), irrespective of applications executed on it.

Both SPEC CPU2006 benchmarks (listed in Table **??**) and PARSEC benchmarks (listed in Table **??**) were chosen for assessing our TAS. It is well known that SPEC benchmarks are intended for assessing CPU of processors, thereby including limited execution parallelism via multithreading and unable to benefit noticeably from executed on a multicore system with more effective thread scheduler (like our TAS), since their execution tends to be single-threaded. By contrast, PARSEC benchmarks are multi-threaded programs aiming at on-chip multiprocessor (and shared-memory system) evaluation. With abundant threads for execution on the multiple cores of the testbed, PARSEC banchmarks are expect to enjoy higher gains under a more effective scheduling mechanism, as will be demonstrated in Section **??** below.

TABLE III: SPEC benchmarks used for evaluation

Integer Benchmarks	
hmmer	Search gene sequence
mcf	Combinatorial optimization
Floating Benchmarks	
milc	Quantum Chromodynamics
libquantum	Physics/General Relativity
namd	Fluid Dynamics

D. SPEC Benchmark Behavior and Discussion

It has been shown in prior work [**?**], [**?**] that significant core-to-core and functional unit-to-functional unit thermal variation occurs on modern processors. Benchmarks from the SPEC CPU2006 suite [**?**], (as listed in Table (**??**)), were used to compare the thermal behavior of CPU-bound workloads executed on the TAS scheduler vs. same workload for the ULE scheduler.

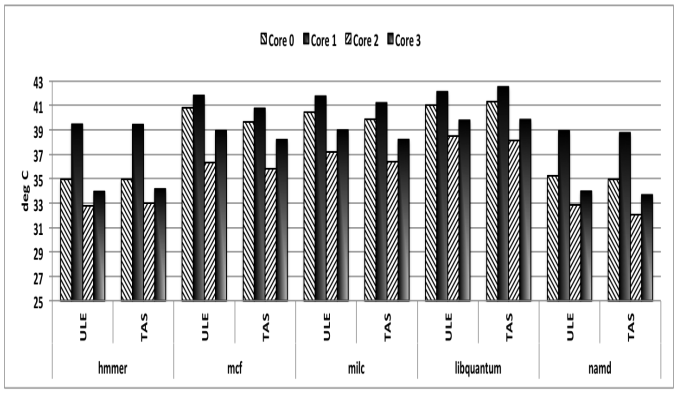

Fig. 4: Average core die temperature under selected SPEC CPU20006 benchmarks for ULE and TAS schedulers.

In Fig. **??**, we see the average core die temperature for each processor in our test system for each of the benchmarks in Table **??**. We see very little temperature variation between the two schedulers in this case as these benchmarks are CPU-bound which means that threads for these processes tend to congregate on the same core due to cache affinity. Thus, the lack of temperature variation as the TAS adjusts its thread selection to minimize the impact in performance for workloads of this type.

1) Execution of SPEC Benchmark Combinations

In this section, we consider the effect of mixed workloads multi-threading across multiple logical cores. We construct mixed workloads from combinations of the SPEC CPU2006 benchmarks that span a range of cases including high-IPC/low-IPC, hot/cold thermal profiles (per [**?**]), and integer/floating-point combinations (Table **??**). A single copy of each benchmark was executed in parallel with the operating system making the decision as to where to execute the workload. It was observed that as the workload progressed, the executed threads again tended to pin themselves to a particular core depending upon thread use of cache and memory.

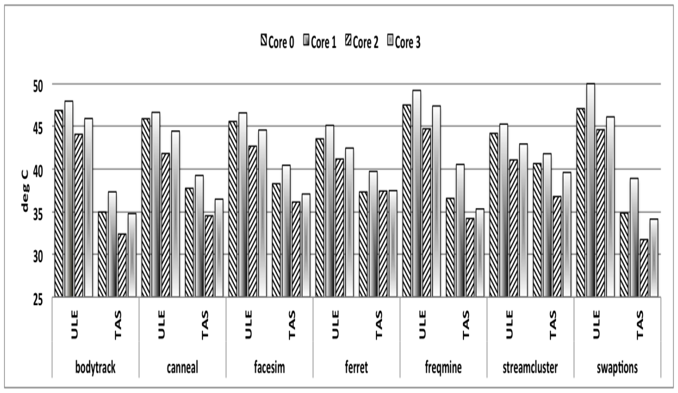

Fig. 5: Comparison of PARSEC benchmark average core die temperatures under ULE and TAS schedulers.

In this experiment, we see temperature improvements between 1.7% and 7.1% (as shown in Table **??**) with a performance impact of between 2.1% and 2.9%. This compares favorably with techniques from prior work such as Core Hopping [**?**], Variation-Aware [**?**], and Predict-and-Act [**?**], with comparable reductions in average core die temperatures and performance impact for similar workloads and processors.

E. PARSEC Benchmark Behavior and Discussion

We evaluate behavior of our scheduling algorithms when executing highly parallel workloads using selected benchmarks from the PARSEC [**?**] suite. Benchmarks in this suite use different combinations of parallel workloads selected from the fields of computer vision, computational finance, enterprise servers, and animation physics as described in Table **??**. The benchmarks were compiled with

TABLE IV: Various SPEC benchmark combinations for concurrent execution

Workload	Thread 0	Thread 1	Thread 2	Thread 3
A	namd	namd	hmmer	mcf
B	milc	milc	namd	hmmer
C	milc	mcf	mcf	hmmer

TABLE V: Reduction in temperature and performance under benchmark combinations

Workload	Core 0	Core 1	Core2	Core 3	Perf. Impact
A	2.8°C	1.0°C	1.7°C	2.0°C	2.1%
B	1.0°C	0.8°C	2.0°C	1.1°C	2.9%
C	3.1°C	3.3°C	3.0°C	2.9°C	2.5%

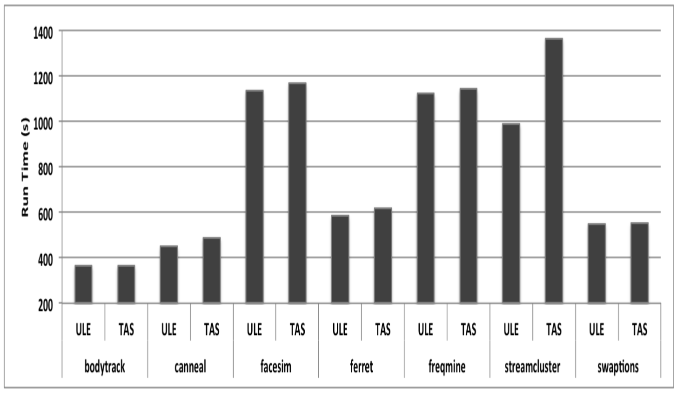

Fig. 6: Comparison of PARSEC benchmark performance under ULE and TAS schedulers.

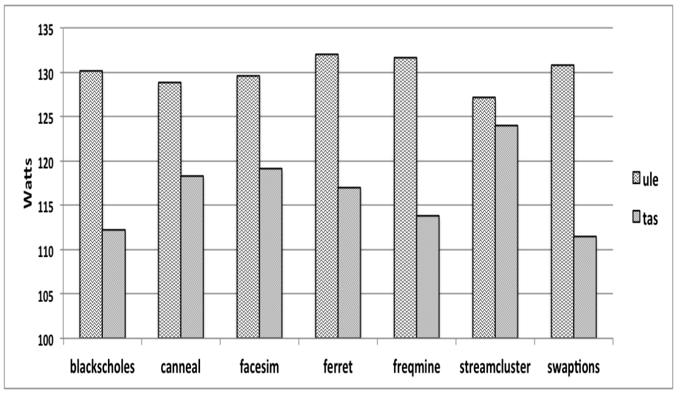

Fig. 7: Comparison of PARSEC benchmark average power consumption under ULE and TAS schedulers.

the POSIX `pthreads` library and executed using the PARSEC `native` input sets.

Three metrics were used to compare the behavior oF the ported PARSEC benchmarks for the original ULE scheduler and our TAS: (a) the average core die temperature (Fig. **??**), (b) benchmark performance (Fig. **??**), and (c) average system power (Fig. **??**). We see improved performance from benchmarks such as `bodytrack` and `swaptions` that utilize smaller working sets and consequently have less need for bigger cache capacity [**?**] with TAS showing core temperature reduction by up to 12°C. with negligible performance degradation. Benchmarks with streaming behavior such as`facesim`, `freqmine`, or `streamcluster` with much larger working sets demonstrate less improvement with TAS, with the average core die temperature lowered by 3-6°C. less than ULE.

In Fig. **??**, we contrast the optimization strategy used by TAS versus other possible optimizations schemes for managing the trade-off between performance and energy. In this figure, we compare TAS performance results from executing the PARSEC `facesim` benchmark as compared to schemes that minimize delay under peak power constraints (DPC) and minimize energy under peak power and delay constraints (EPDC). DPC minimizes the delay of an execution interval so that the maximum power within an execution interval never exceeds a limit while EPDC schemes seek to minimize energy while operating a system within a power budget with a limit on the workload performance impact. In this case, the optimizations used by TAS outperforms both DPC and EPDC for the `facesim` benchmark by 4% to 6% as reported in prior work [**?**].

TABLE VI: PARSEC benchmarks used in evaluation

bodytrack	Computer vision image tracking application
canneal	Simulated annealing chip routing cost computation
facesim	Physical simulation of facial behavior
ferret	Content-based similarity search
freqmine	Simulate FP-growth method for Frequent Itemset Mining
streamcluster	Online clustering algorithm for data mining
swaptions	Simulated pricing of portfolio options

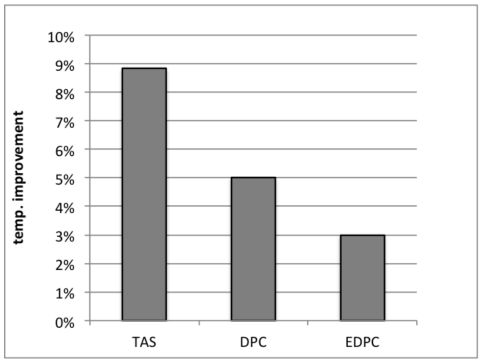

Fig. 8: Average percentage reduction in core die temperature for the `facesim` benchmark under various optimization strategies.

VI. Conclusion

Dense servers pose power and thermal challenges, which often cannot be addressed satisfactorily by conventional DVFS and DTM mechanisms, especially under heavy workloads common for high-performance systems. We have investigated into thermal-aware scheduling to deal with such challenges, capable of managing system energy consumption within a given power and thread envelope effectively. As opposed to prior work aiming to bound temperatures below critical thresholds, our proposed scheduler considers how to dispatch high workloads in the high-performance multi-core system for die temperature management across all cores. It is based on the thermal Chaotic Attractor Predictors (tCAPs) we develop to guide thread selection and load balancing, taking into account key thermal indicators and system performance metrics for preventing DTM instances proactively. The proposed tCAP-oriented scheduling (dubbed the TAS scheduler) has been implemented to replace the original scheduler of the FreeBSD operating system (called the ULE scheduler) for evaluation on a testbed server under benchmarks from the SPEC CPU2006 and PARSEC suites. Experimental results demonstrate that our TAS scheduler can lower the mean on-die core temperature by up to 12°C under PARSEC benchmarks and by up to 3.3°C under mixes of

SPEC benchmarks for concurrent execution, while exhibiting negligible performance degradation, in comparison to the ULE scheduler. When compared with a recent energy-aware scheduling technique reported to attain core temperature reduction by 4°C upon executing the parallel benchmark of Charm++ on an Intel Xeon 5520 4-core processor [**?**], our TAS clearly enjoys better thermal reduction under multi-threaded execution.