System-level, Thermal-aware, Fully-loaded Process Scheduling

Dong Li, Hung-Ching Chang, Hari K. Pyla, Kirk W. Cameron
Department of Computer Science, Virginia Tech
{lid, hcchang , harip, cameron} @ vt.edu

Abstract

Processor power consumption produces significant heat and can result in higher average operating temperatures. High operating temperatures can lead to reduced reliability and at times thermal emergencies. Previous thermal-aware techniques use Dynamic Voltage and Frequency Scaling (DVFS) or multithreaded or multicore process migration to reduce thermals. However, these methods do not gracefully handle scenarios where processors are fully loaded, i.e. there are no free threads or cores for process scheduling. We propose techniques to reduce processor temperature when processors are fully loaded. We use system-level compiler support and dynamic runtime instrumentation to identify the relative thermal intensity of processes. We implement a thermal-aware process scheduling algorithm that reduces processor thermals while maintaining application throughput. We favor "cool" processes by reducing time slice allocations for "hot" processes. Results indicate that our thermal-aware scheduling can reduce processor thermals by up to 3 degrees Celsius with little to no loss in application throughput.

1. Introduction

Power consumption is now a critical design constraint in high-end system design. Increased power consumption produces additional heat which must be dissipated by complex cooling systems. It can result in higher average operating temperatures which decrease the reliability of microelectronics. The Arrhenius equation[1] states a temperature increase of 10 degrees Celsius results in reliability decrease of an electronic device by 50 percent. In a compute server cluster this translates to a shorter average life span for each electronic device and a shorter mean-time-between-failure. Therefore, thermal-aware design is playing a more and more important role for sustaining high-performance computing.

Recent thermal management techniques [2-6] include scheduling of power modes such as DVFS and scheduling threads and cores in SMTs and CMPs respectively to reduce thermal hotspots. Unfortunately, DVFS techniques may not provide enough granularity for thermal control. Currently, architectures support only lock-step frequency changes limiting the amount of control possible on real systems. DVFS techniques also can hurt performance and are primarily a last resort to protect the processor from thermal emergencies. Use of DVFS to reduce thermals in typical situations may unfairly penalize "cooler" running processes by reducing their performance. Process migration techniques can provide some relief, but there are currently no solutions that suggest process migration schedules to improve thermals when all cores or threads are active. Nevertheless, in previous work we have created tools[7] that identify hot and cool running threads. These findings indicate scheduling based on temperature in cases when all resources are being used could reduce temperature. The challenge is to ensure the techniques used do not impact performance negatively.

This paper proposes a thermal control framework based on process scheduling (process priority setting) to address these challenges. Ours is an architecture independent solution, controlling temperature at fine (process) granularity. Our techniques operate at the user application level and do not require any modifications to the operating system kernel. Typical research in this area uses simulation for validation such as Turandot[8], PowerTimer[9] and HotSpot[10]. In these cases, changing system design at any level means redesign and revalidation of the thermal models. Simulations of this type are also impractical for studying thermal optimization techniques in large high-end clusters. We verify our techniques on a real system running with an AMD 64bit Athlon processor. More specifically, our approach uses profile-driven techniques to obtain process information on resource intensity from hardware performance counters. We also gather data from temperature sensors. We combine this information to identify the relative

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

thermal intensity of a process. Whenever processor temperature is above a user specified value, we trigger our process scheduling algorithm to reset process priority to favor "cool" processes.

The rest of this paper is organized as follows. Section 2 discusses related work in the area of temperature aware computing. Section 3 gives background information. In section 4, we present the complete thermal control framework. Section 5 presents our experimental analysis and results. Finally, in section 6 we conclude the paper.

2. Related Work

Many have explored the use of DVFS to maintain temperature thresholds and/or reduce temperature. Since temperature has strong relationship to processor resource usage and DVFS is a fast way to reduce the thermal output of all resources, DVFS is very efficient for temperature control. Pyla, et al.,[5] showed the effects of DVFS on real system thermals and designed a runtime PID controller for CPU frequency setting. With the supports of the compiler and thermal sensors, their work provided fine-grained thermal profiling of parallel scientific applications. Isci, et al.,[11] presented a system for predicting phases of applications at runtime using performance counters. It uses a global phase history table predictor leveraged from a common branch predictor technique. These runtime phase predictions are used to guide DVFS as the underlying dynamic management technique. Donald, et al.,[3] categorized previous thermal management techniques and claimed that the best performing thermal control combination includes both control-theoretic distributed DVFS and a sensor-based migration policy. However DVFS, including distributed DVFS which operates on the core scale, although it is very effective in decreasing temperature, may not provide enough granularity for thermal management since most current systems do not support independent frequency scaling of cores or threads. Our methods operate at finer (process) granularities by treating scheduling of cores independently and avoiding the use of DVFS.

Recent research leverages CMPs and SMT cores for adaptive thermal control. Powell et al.,[4] discuss thread migration as an effective scheme to avoid overheating of cores. They achieve this by assigning workloads to free SMT contexts on alternative cores, leveraging availability of SMT contexts on alternate CMP cores to maintain throughput while allowing overheated cores to cool. However this method is not effective when there are no free SMT contexts on all cores. Our work doesn't depend on CMP and SMT architectures and can handle scenarios where all cores or threads are in use.

Donald et al.,[2] claimed that when multiple heterogeneous programs are available in the workload, thermal-aware instruction issue policies provide a significant power-performance benefit. Their work selectively manages process execution when there are opportunities for adaptively counteracting and preventing hot spots assuming hardware support. However if programs are homogeneous, their methods are ineffective. Our work can identify resource usage intensity regardless of process mix and intervenes through the process priority mechanism of the operating system. Li et al, [12] explain that the mechanisms by which SMT and CMP heat up are quite different. SMT heat up is primarily caused by localized heating while CMP heat up is mainly caused by the global impact of increased energy output. They conclude non-DVFS localized thermal-management can outperform DVFS for SMT. Our approach is architecture-independent and thus can be applied in both SMT and CMP.

Bellosa et al,[13] proposed a modified process scheduler which can allocate CPU time slices according to the power consumption of each task to the current temperature level of the processor. In essence, both their methods and our methods do thermal management by controlling CPU time slice allocation. Their thermal model is based on a power model and can ignore localized hotspots. On different platform a power model must be derived and verified. Since we use a combination of hardware counters and thermal sensors our methods are more portable and thus don't require model conversion. We additionally use the collected data to identify the relative resource intensity of processes. This is based on the reasonable assumption that resource intensity has strong relationship with the temperature.

3. Background

The process scheduling mechanism in a multitasking operating system is used to select which process to run next. It can be viewed as the subsystem of the kernel that divides the finite resources of processor time between the running processes on a system. By deciding what process can run, the scheduler is responsible for best utilizing the system. In Section 3.1, we describe the process scheduling mechanism in the Linux 2.6 kernel on which our framework is based. We note that many operating systems use similar scheduling policies. Thus, it is not be difficult to port our algorithm onto systems. In

addition this work uses temperature sensors to capture thermal status. We describe the sensors in section 3.2.

3.1. Linux Process Scheduling

Linux uses a priority-based scheduling algorithm. The idea is to rank processes based on their need for processor time. Processes with a higher priority run before those with a lower priority, whereas processes with the same priority are scheduled round-robin. On Linux, processes with a higher priority also receive a longer time slice. The process with time slices remaining and the highest priority always runs. Both the user and the system may set a process's priority to influence the scheduling behavior of the system.

Processes have an initial priority that is called the nice value. This value ranges from -20 to +19 with a default of zero. +19 is the lowest and -20 is the highest priority. The variable is also called the static priority because it does not change from what the user specifies. The process scheduler, in turn, bases its decisions on the dynamic priority. The dynamic priority is calculated as a function of the static priority and the task's interactivity. The scheduler computes a bonus or penalty in the range -5 to +5 based on the interactivity of the task.

Our thermal management system sets process nice values based on process relative resource intensity. In this way we can reduce time slice allocation for "hot" processes when the temperature is high. Effectively, we are interleaving the scheduling of cooler processes to reduce the heat created by high intensity hot processes.

3.2. Temperature Sensors

We use AMD K8 digital temperature sensors embedded in the AMD Athlon processor to report core CPU temperature. The sensor is digital, which means it does not rely on an external circuit located on the motherboard to report temperature. Its value is stored in a special register in the processor so software can access and read it. This eliminates any inaccuracy that can be caused by external motherboard circuits. Many processors have this kind of sensor, including Intel Core Duo, Core Solo, Pentium E1000 series, Xeon series etc. Even for processors that don't have thermal sensors,[14] describes repeatable accurate methodologies for measuring the processor-die temperature.

Lee etc. claimed there is temperature differential between a hot spot and a region of interest based on their distance in the chip and processor packaging information[15], i.e. the placement of sensors play an important role in reporting the temperature measurement of hot spot. We assume the placement of K8 thermal sensor in the chip is accurate. Our techniques effectively reduce the measured temperature. Hence, despite this assumption our results show promise for CPU heat reduction. In the worst case, our measurements would underestimate the temperature reduction.

4. Design and Implementation

Our thermal management framework consists of a process profiler and a control daemon shown in Figure1. The profiler is a shared library attached to application processes to record resource access counts. The control daemon is a system service which collects process information and reads the temperature sensors. It is also used to makes process scheduling decisions. We use Performance Application Programming Interface (PAPI) [16] to record resource access counts. The PAPI events have direct relationship with the CPU functional units. So the event numbers reflect access intensity.

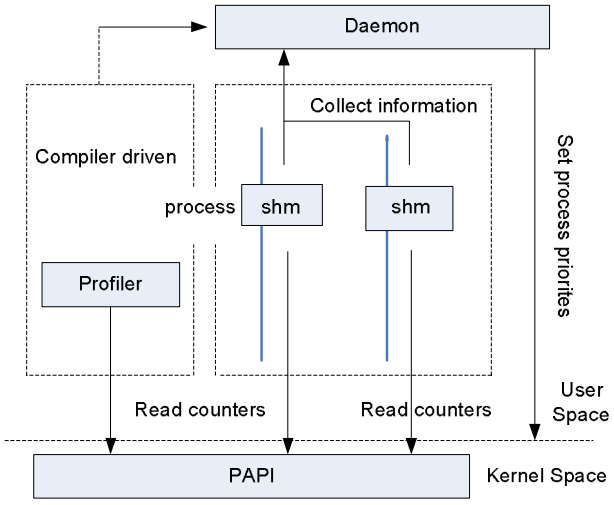


Figure1. Our thermal management system design

The profiler instruments the process to collect resource access information. Its functionality includes: (1) initializing the PAPI library; (2) managing shared memory segment and advertising segment ID for inter-process communication (IPC); (3) periodically reading PAPI event number and updating shared memory area.

The profiler initializes the PAPI library when it is loaded for the first time. It then reads the configuration file to get the event names and adds them into the event set. The configuration file contains the *event name* registered by the user for profiling. The profiler

uses the shared memory segment to export event numbers to the control daemon. Shared memory is the fastest form of IPC. It doesn't require a system call or entry to the kernel and avoids copying data unnecessarily. However to access the shared memory, its segment ID is needed. We define this directory as "*segment ID store*", where the profiler creates an empty file named with the attaching process's ID. Then a new unique process identity can be formed by converting this file pathname to an IPC key. The profiler uses this key to create a shared memory segment ID. The daemon scans the segment ID store to find application processes and form the keys in the same way as profilers. The same segment ID can then be produced by the daemon to access shared memory. The profiler periodically outputs the event numbers into shared memory. The time period here is set in terms of process virtual time instead of wall clock time. This guarantees the events number we record actually reflect the resource intensity of a process even when the process is set down to a low priority. This also avoids the delay delivery of timer signal under heavy system loading. To synchronize accesses of shared memory and avoid the overhead of locking, we use version control. The daemon can detect memory updates by looking for changes to version number. If the version number is not changed, the daemon uses the old data obtained in the previous period as an estimate. Whenever the application process finishes, the profiler removes its corresponding file from the segment ID store. The daemon in turn notices this change and deletes the memory segment. So the segment ID store is actually a registry. The daemon scans it to maintain the process membership for thermal management.

We leverage the constructor/destructor attributes of the GNU compiler to attach profilers. These attributes cause profiler functions to be called before and after the execution of the program. This is easy to implement. However, such a technique is compiler dependent and requires re-linking applications. Alternatively, our system uses Dyninst[17] to attach to processes. Dyninst is based on the idea of dynamic instrumentation[18] and permits the insertion of code at runtime into a program without the need of re-compilation or re-linkage. Our system includes a *mutator* program based on Dyninst, it takes the profiler code snippet and inserts it at the starting point of the applications. This method is completely application-independent. However installing and developing using Dyninst is not an easy task and requires extra code development (*mutator* program). Our framework supports both these approaches to instrument an application. We tried to design the profiler with low overhead and avoid performance loss in application processes. We choose to sample the performance counters once every second, we believe this interval is consistent enough to reflect the resource variance. Table 1 illustrates the overheads of various NAS serial benchmarks.

Table 1. Performance overhead of benchmarks with/without the profiler.

Benchmark	Time without profiler (s)	Time with profiler (s)	Performance loss
CG.B	246.04	250.53	1.82%
LU.A	181.77	184.02	1.24%
SP.A	212.46	213.72	0.593%
EP.B	131.39	131.81	0.320%

The daemon (figure1) controls thermal management. It also periodically scans the *segment ID store* directory for any newly registered processes and also cleans the memory segment for a terminated process. Whenever a new process is found, the daemon records its initial priority and creates its memory segment ID. The daemon at regular intervals, (wall clock time) reads the thermal sensors and compares it with the temperature threshold set by the user. The process scheduling routine is triggered whenever the temperature is above the threshold. The sampling rate for thermal sensors should be longer than the time it takes the kernel to update the sensor register and also should be short enough to reflect temperature variance. As a result, based on our experiments we choose a sampling rate of once per second.

The process scheduling algorithm is intuitive. The scheduler first reads the shared memory the exported by the currently registered processes. This is done to figure out the resource consumption (event numbers) within a time interval. The most resource intensive process is given a lower static priority than its current priority. The difference in each priority level is chosen as 5. Since the Linux scheduler computes bonus or penalty for the process dynamic priority in the range -5 to +5, we offset the effects of the Linux scheduler by lowering the priority by 5. This is repeated until the following two conditions are satisfied. (1) *The temperature is below the threshold.* In which case, the processes static priorities are immediately restored their original values. In general, we decrement the priorities of processes gradually by steps of 5 and reset them quickly to avoid any performance loss. (2) *All registered processes reach their least priority possible.* In such a scenario there is no room to further decrement the process priority. In which case, we set process priority to the original value and reshuffle scheduling. This is important for the following reason:

When all processes are resource intensive and thus have a lowest priority possible, over a period of time, they tend to use resources much less frequently. We should favor such processes by assigning a higher priority. However according to (1), unless the temperature is below the threshold, we do not raise the process priority. So our thermal control fails. But by reshuffling scheduling, we expose more opportunities for process scheduling and avoid this extreme case of failure.

```
1.  Initialization:
2.      Initialize the process set as empty;
3.      Read the configuration file to get temperature thresholds and
4.      process priority range;
5.      Record the current CPU frequency f;
6.  Iteration:
7.      Scan the segment ID store to register new processes and
8.      remove finished processes from the process set;
9.      Read the thermal sensor;
10.     If ( temperature > critical temperature ) {
11.       If ( temperature > emergency temperature )
12.         Set CPU freq to the lower by DVFS;
13.       else {
14.         Read event numbers from the share memory of each
15.         registered process;
16.         Find the process with the largest event number and set its
17.         priority 5 less;
18.         If( all processes are in the lowest priority )
19.             Set priorities to the original value;
20.       }
21.     }
22.     else{
23.       Set all processes' priority to the original value;
24.       Set CPU freq to the original value f;
25.     }
26.     Suspend the daemon for a time interval T;
```

Figure 2. The algorithm for the daemon

Since the "hot" process's priority is lowered gradually, our thermal management may not react quickly enough to avoid thermal emergency. So we use the DVFS as our backup policy to fall upon in order to guarantee prevention of a thermal emergency. The complete scheduling algorithm is shown in the Figure2.

5. Evaluation

In this section, we examine the benefits of our temperature-aware process scheduling. Depending on the event type observed by the profiler, the daemon can commit the scheduling for homogenous processes and heterogeneous processes. In our system we define processes to be "*homogenous*" when their thermal properties can be characterized by repeated access to the same resource, such as a floating point unit or a fixed-point execution unit etc. For the homogeneous processes, we record PAPI event number corresponding to the resource they access most intensively. For the heterogeneous processes, we compute and compare IPC based on the PAPI events. When IPC is large enough (>0.5), it generates a rough estimate of power dissipation[19] which could provide an estimate to temperature variance. So the scheduling for homogenous processes actually focuses on the thermals of individual functional units, while the scheduling for heterogeneous processes focuses on the thermals of the whole chip.

Table 2. Benchmarks used for evaluation.

Group Label	App	Int/FP	Ex-IPC	Homo /hetero
A	cg+sixtrack	FP+FP	L+H	Homo
B	sixtrack+art +apsi	FP+FP+ FP	H+L+L	Homo
C	art+mesa	FP+Int	L+H	Hetero
D	mcf+art +gzip	Int+FP+ Int	L+L+H	Hetero

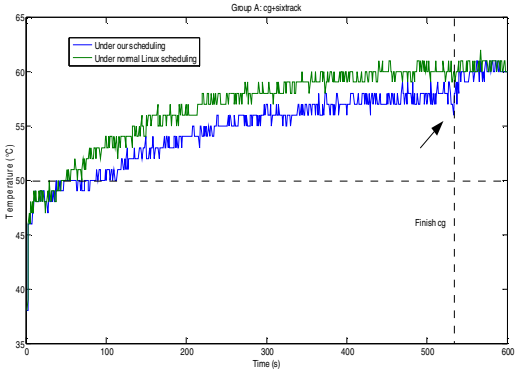
We ran our experiments on a HP DX5150 MT PC with AMD Athlon64 4000+ 2.4GHz CPU (single core without SMT) with 1GB memory and running Linux (kernel 2.6.19) kernel. We did not trigger DVFS in the case of a thermal emergency and CPU fan speed was set to a constant value, this is done to minimize the impact of these controls and understand the true effects of process priorities on thermals. The benchmarks in the experiments are selected from the SPEC CPU 2000 suite and the NAS parallel benchmark shown in the table 2. They are compiled with the default settings using gcc (version 3.4.6) for C programs and gfortran (version 4.1.0) for Fortran programs. We categorize benchmarks into two categories: *high* and *low* based on their ex-IPC. High ex-IPC reflects high core activity and indicates a potential high temperature and vice-versa. Before running any benchmark we allow the computer to sit idle briefly and confirm that it has reached its idle operating temperature. The temperature threshold for enacting thermal control is set to 50°C. Modern commercial microprocessors tend to list maximum allowable operating temperatures in the range of 70-90 °C[20]. Depending on the position of thermal sensors, the overall chip temperature reported by the thermal sensors could be 30°C less than the temperature of a hot spot[2]. The temperature threshold should reflect this difference. In our platform we don't know the exact position of thermal sensors, but we feel 50°C is a reasonable choice. For the *homogenous* process groups which are floating point intensive in our experiments, we record the event number for

floating point instructions, which in turn correlates to the floating point register file access, one of the hottest chip portions.

Table 3. Execution times for various groups using our system (Time-1) and the default Linux scheduler (Time-2).

Group	Benchmarks	Time 1(s)	Time 2(s)
Group A	CG	534	981
	Sixtrack	1234	1194
Group B	Art	274	387
	Apsi	852	1497
	Sixtrack	1556	1552
Group C	Art	154	261
	Mesa	302	303
Group D	Mcf	468	632
	Art	312	366
	Gzip	639	525

Figure 3 reveals that when the processor is fully-loaded the processor temperature is lower under our scheduling than under normal Linux scheduling. The differences of time for finishing benchmark tasks between our scheduling and Linux scheduling is within 4% thus guarantying the system throughput. There are several interesting facts worth noticing. (1) Under our scheduling, "cool" processes finish earlier than under normal Linux scheduling (shown in the table 3). This is due to their relative high priority. After they finish, the "hot" processes are allocated more time slices and thus can run faster. This helps in compensating the amount of time they lost when the "cool" process were running. So in our case there is not a significant difference in terms of the time between running under our scheduling and under normal Linux scheduling. (2) When the "cool" processes are finished, the temperature rises up much quickly. These are shown by the arrow in the figures. At this point, the "hot" process gets more time slices to execute and therefore burns the processor up. (3) Also, at the point when all



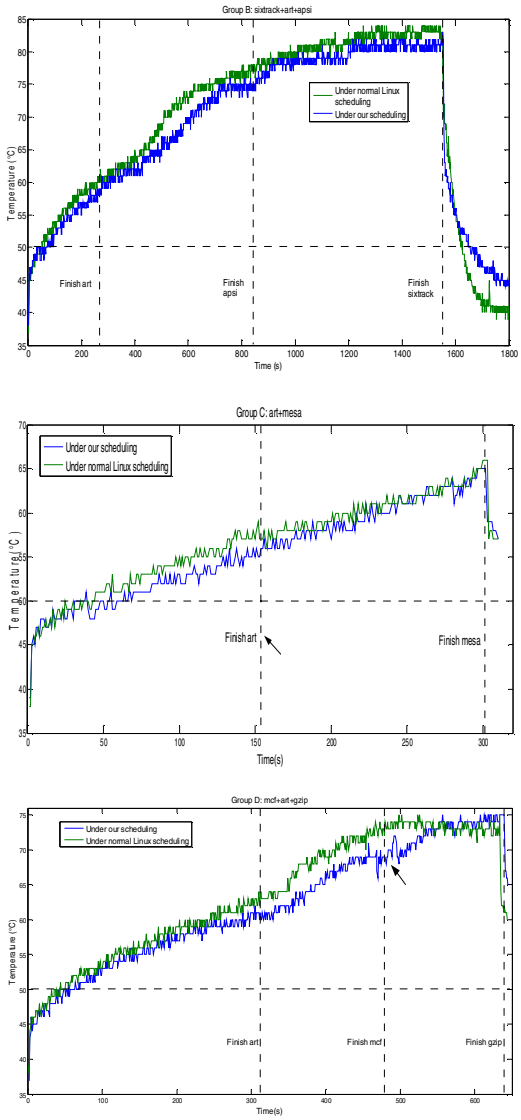


Figure 3: Comparison of our scheduling scheme with default Linux scheduler. The dotted line refers to the finish time under our scheduling.

benchmarks finish, the final temperature under our scheduling scheme could be different from the one under the default case. For example, in groups B and C, our temperature is lower, while in other cases it is higher. Combining (2) and (3), we conclude that given enough "cool" processes to schedule it is possible to achieve lower temperatures.

6. Conclusion and Future Work

This paper proposes a thermal management framework orthogonal to previous thermal-aware research. It targets process scheduling when there are no free simultaneously-multithreaded contexts on all

cores. By leveraging compiler or dynamic runtime instrumentation we attach a profiler to an application collecting process information (resource intensity). The process with high resource intensity is assumed to be "hot" and leads to high temperature. According to the process information we distribute process priority of all registered processes along the allowed range. The "hot" process is assigned relatively low priority. In this way we indirectly reduce time slice allocation for "hot" process and thus cool the processor when it is fully loaded.

Although our work effectively reduces temperature, it doesn't replace previous methods. Our techniques target scenarios common to high-performance computing where processors are fully loaded, i.e. there are no free hardware threads or cores available. In addition, as temperature emergencies could happen at the scale of millisecond, process scheduling may not be fast enough to react against temperature variance. In the future we hope to integrate our techniques with other scheduling techniques and DVFS techniques to improve temperature reduction under various conditions.

7. References

[1] A. McNaught, " Compendium of Chemical Terminology: IUPAC recommendations," 2nd ed: Oxford, 1997.

[2] J. Donald and M. Martonosi, "Leveraging Simultaneous Multithreading for Adaptive Thermal Control," in *2nd workshop on Temperature-Aware Computer Systems(TACS-2)*, 2005.

[3] J. Donald and M. Martonosi, "Techniques for Multicore Thermal Management: Classification and New Exploration," in *33rd International Symposium on Computer Architecture (ISCA-33)*, 2006.

[4] M. D. Powell, M. Gomaa, and T. N. Vijaykumar, "Heat-and-Run: Leveraging SMT and CMP to Manage Power Density through Operating System," in *International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, Boston, MA, 2004.

[5] H. K. Pyla, D. Li, and K. W. Cameron, "Poster: Thermal-aware High-performance Computing Using TEMPEST," in *19th IEEE/ACM International Conference on High Performance Computing and Communications (SC07)*, Reno, NV, 2007.

[6] A. Snavely and D. M. Tullsen, "Symbiotic Job scheduling for a Simultaneous Multithreading Processor," in *International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2000, pp. 234-244.

[7] H. K. Pyla, "Tempest: A Framework for High Performance Thermal-Aware Distributed Computing," in *Computer Science*. vol. Masters in Computer Science and Applications Blacksburg: Virginia, 2007, p. 79.

[8] M. Moudgill, J.-D. Wellman, and J. H. Moreno, "Environment for PowerPC microarchitecture exploration," *IEEE Micro,* vol. 19, pp. 15-25, 1999.

[9] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook, "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors," *IEEE Micro,* 2000.

[10] K. Skadron, T. Abdelzaher, and M. R. Stan, "Control-Theoretic Techniques and Thermal-RC Modeling for Accurate and Localized Dynamic Thermal Management," in *Eighth International Symposium on High-Performance Computer Architecture*, 2002, pp. 17-28.

[11] C. Isci, G. Contreras, and M. Martonosi, "Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management," in *39th ACM/IEEE International Symposium on Microarchitecture (MICRO-39)*, 2006.

[12] Y. Li, D. Brooksz, Z. Huyy, and K. Skadron, "Performance, Energy, and Thermal Considerations for SMT and CMP Architectures," in *11th International Symposium on High-Performance Computer Architecture (HPCA-11)*, 2005, pp. 71-82.

[13] F. Bellosa, S. Kellner, M. Waitz, and A. Weissel, "Event-Driven Energy Accounting for Dynamic Thermal Management," in *Proceedings of the Workshop on Compilers and Operating Systems for Low Power*, 2003.

[14] AMD, "Methodologies for Measuring Temperature on AMD Athlon and AMD Duron Processors." vol. 2008.

[15] K.-J. Lee, K. Skadron, and W. Huang, "Analytical Model for Sensor Placement on Microprocessors," in *International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*. 2005, pp. 24-27.

[16] "Performance Application Programming Interface," p. http://icl.cs.utk.edu/papi/.

[17] "An Application Program Interface (API) for Runtime Code Generation."

[18] J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools," in *Scalable High Performance Computing Conference*, 1994, pp. 841-850.

[19] W. L. Bircher, J. Law, M. Valluri, and L. K. John, "Effective Use of Performance Monitoring Counters for Run-Time Prediction of Power", 2004.

[20] "CPU Maximum Operating Temperatures."