Tempest: A Framework for High Performance Thermal-Aware Distributed Computing

Hari Krishna Pyla

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Dr. Srinidhi Varadarajan (Chair)

Dr. Kirk W. Cameron

Dr. Calvin J. Ribbens

Dr. Naren Ramakrishnan

May 18, 2007
Blacksburg, Virginia

Keywords: High Performance Computing, Thermal Profilers, Thermal Controllers.

Tempest: A Framework for High Performance Thermal-Aware Distributed Computing

Hari Krishna Pyla

ABSTRACT

Compute clusters are consuming more power at higher densities than ever before. This results in increased thermal dissipation, the need for powerful cooling systems, and ultimately a reduction in system reliability as temperatures increase. Over the past several years, the research community has reacted to this problem by producing software tools such as HotSpot and Mercury to estimate system thermal characteristics and validate thermal-management techniques. While these tools are flexible and useful, they suffer several limitations: for the average user such simulation tools can be cumbersome to use, these tools may take significant time and expertise to port to different systems. Further, such tools produce significant detail and accuracy at the expense of execution time enough to prohibit iterative testing. We propose a fast, easy to use, accurate, portable, software framework called Tempest (for *temp*erature *est*imator) that leverages emergent thermal sensors to enable user profiling, evaluating, and reducing the thermal characteristics of systems and applications.

In this thesis, we illustrate the use of Tempest to analyze the thermal effects of various parallel benchmarks in clusters. We also show how users can analyze the effects of thermal optimizations on cluster applications. Dynamic Voltage and Frequency Scaling (DVFS) reduces the power consumption of high-performance clusters by reducing processor voltage during periods of low utilization. We designed Tempest to measure the runtime effects of processor frequency on thermals. Our experiments indicate HPC workload characteristics greatly impact the effects of DVFS on temperature. We propose a thermal-aware DVFS scheduling approach that proactively controls processor voltage across a cluster by evaluating and predicting trends in processor temperature. We identify

approaches that can maintain temperature thresholds and reduce temperature with minimal impact on performance. Our results indicate that proactive, temperature-aware scheduling of DVFS can reduce cluster-wide processor thermals by more than 10 degrees Celsius, the threshold for improving electronic reliability by 50%.

To my parents and my brother.
To my friends and family.
To the thirty-two Hokies we
lost on 04/16/07.

Acknowledgements

I take this opportunity to thank all the people who helped me in various ways throughout my graduate study here at Virginia Tech.

My advisor, Dr. Srinidhi Varadarajan, who has been a friend, philosopher and a great inspiration to me. He advised me and funded me during various stages of this work.

Dr. Kirk W. Cameron, for helping me with his able guidance throughout this thesis and for his constant support while publishing results.

Dr. Calvin J. Ribbens, for funding me during my first two semesters and his help with various official issues, his guidance, and his mentorship.

Dr. Naren Ramakrishnan, for his suggestions, support, encouragement and words of wisdom and advice.

My Masters committee- Dr. Kirk W. Cameron, Dr. Calvin J. Ribbens and Dr. Naren Ramakrishnan.

My brother Pardha S. Pyla for his inspiration and bearing with me.

My friend and inspiration Dr. Joy Mukherjee.

The GNU Compiler collection (GCC) mailing lists.

The lm_sensors mailing lists.

The staff at the Department of Computer Science. In particular Ginger Clayton, Melanie Darden, Rachel Smith, and Jessie Eaves.

My friends who made my tea time a mix of fun and relaxation: Dr. Omprakash Seresta, Bharath Ramesh, Patrick Liesveld, Vedvyas Duggirala, Lee Smith, Rajesh Sudarsan, Mehmet Belgin, Satish Tadepalli, Pilsung Kang, Abhijit Deodhar, Chreston Miller, Ankur Shah, Pavan Konanki and Dong Li.

My Colleagues at the Computing Systems Research Laboratory (CSRL): Dr. Joy Mukherjee, Bharath Ramesh, Craig Bergstrom, Pattrick Liesveld, Vedvyas Duggirala, Lee Smith, Pilsung Kang and Joe Ruscio.

My Colleagues at the Scalable Performance Evaluation Laboratory (SCAPE): Dr. Xizhou Feng, Rong Ge, Dong Li, Song Li and Matt Tolentino.

Hari Krishna Pyla

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The power consumption of large scale clusters is now a critical design constraint in high-performance clusters. The power consumption of servers, the building blocks of high-end clusters, is increasing for two primary reasons. First, the pursuit of Moore's law has led to devices that contain large numbers of transistors at higher densities. Second, the pursuit of performance has led to systems (server clusters and data centers) with large numbers of power-hungry components in close proximity. As a result, the US Environmental Protection Agency recently announced its intention to create and reward EnergyStar ratings for server designs, a program popularized for appliances and monitors in the last 25 years[1].

Increased power consumption and system densities have multiple side effects. Operation costs increase for higher powered clusters. Meanwhile power consumption produces additional heat which must be dissipated by complex cooling systems and can result in higher average operating temperatures which decrease the reliability of microelectronics. For example, the Arrhenius equation 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 (MTBF).

To curb the effects of rising thermal temperatures in compute servers, the research community has introduced tools to enable design and validation of thermal management

techniques that reduce heat dissipation. Light-weight tools use direct thermal sensor measurement, emphasizing speed and low overhead. These tools [2, 3] are primarily designed to provide fast access to real time temperature information for use in thermal management policies. Such techniques are particularly useful for making runtime steering decisions to reduce heat dissipation. Since the focus is to provide temperature information quickly, the profiling aspects of these direct measurement techniques are limited[1].

Heavy-weight tools use software thermal models of system hardware, emphasizing flexibility and accuracy. These simulation tools [4, 5] [6-10] [11] are primarily designed to provide a means of estimating the thermals of proposed hardware configurations. Profiling data from simulation is extremely detailed and while thermal simulations can be used to profile and study the effects of temperature-aware designs, such use is somewhat prohibitive. Thermal simulation of a single processor or system may require a team of experts from several disciplines including material science, mechanical and electrical engineering, computational fluid dynamics, and computer systems and architecture[12]. Even small system changes may require redesign and revalidation of the thermal model. Additionally, the time necessary to obtain simulated data is often orders of magnitude slower than runtime sensor data obtained from a real system.

Heavy-weight tools provide detail at the expense of speed while light-weight tools provide speed at the expense of detail. This leaves a noticeable gap between the two extremes of thermal profiling and analysis. For instance, which tool is best to answer the following fundamental parallel application-related questions (among the first asked by a user interested in the effects of thermal management)?

1. What parts of my parallel application will benefit from thermal management techniques?

[1] While the steering and control techniques themselves are often quite useful, we are speaking to the lack of insight provided by the raw temperature samples being used.

2. Where do I start optimizing my parallel application to reduce thermals?
3. Are the thermal properties of my application similar across machines in a cluster?
4. What and where are the performance effects of thermal optimizations on my application?
5. How do I regulate thermals within the threshold limits and yet maintain performance?

Clearly, light-weight tools may provide the portability necessary to gather thermal profiling data, but they are limited to temperature measurements and do not provide the level of detail needed to answer questions 1, 2, 4 and 5. Heavy-weight tools can provide the detail necessary, but many thermal simulators do not run native system software which may lead to inaccuracies. Furthermore, porting such tools to several systems quickly is also questionable due to thermal model development requirements. Also the time needed for iterative experimentation will probably be prohibitive as well. Thus, heavy-weight tools may be impracticably slow, cumbersome or inaccurate for answering questions 1-5.

Recent research has explored the use of variable power modes to reduce the power consumption of high-performance clusters. Most techniques schedule low power processor frequencies during slack periods in high-performance codes when performance is not affected significantly by processor clock rate [13, 14]. These techniques offer, at times significant power and energy savings, and authors typically claim such reductions directly affect temperature and reliability.

Unfortunately, to the best of our knowledge, no studies of the direct effects of power modes on system temperature have been accomplished. There have been simulation-based studies [4, 15] [6-8, 16, 17] [11] of the effects of power modes on microarchitecture designs, particularly as techniques to avoid thermal emergencies. There are also frameworks for simulating the effects of air flow changes (e.g. broken fan) and variations in component heat dissipation. However, all such techniques are currently simulated and based on complicated thermodynamic models of materials (e.g. chip,

package, heat sink) or airflow between components in a server. Determining the thermodynamic models alone requires experts from several disciplines. Changing system design at any level means redesign and revalidation of the thermal models. Simulations of this type are impractical for studying thermal optimization techniques in large high-end clusters.

In our work, we show that thermal on-chip sensors provide repeatable results for several machines in typical system room operating conditions contrary to some suppositions [12]. While we did encounter sensors that did not work as expected, these were the exception, not the rule, and we believe the technologies will improve over time. We also found no indication of gross inaccuracies when comparing sensor measurements to external multimeter measurements. Hence, we believe that temperature-aware high-performance computing techniques that leverage data from on-board sensors will improve system managers' and users' abilities to reduce the thermal properties of systems and applications.

We advocate the use of thermal sensors to monitor the temperature of various critical components in a high-end cluster. We focus on the use of sensors embedded in the microprocessor. Such sensors provide accurate, real time core CPU temperature data. In this thesis, we propose an infrastructure to gather data from these sensors and profile-driven techniques to control the thermal characteristics of an application at runtime. Though power and energy are important considerations for high-end clusters, in this work we focus on empowering users with the ability to set thermal thresholds in a high-end cluster that maximize performance under temperature constraints.

We propose a *middle-weight* thermal profiling, analysis, and controlling framework that provides the detail necessary to answer such important questions quickly while simultaneously maintaining portability and usability. We call our framework ***temp****erature* ***est****imator*, or ***Tempest*** for short.

Tempest offers its services in the form of shared libraries and provides API to obtain profiles, analysis and maintain temperature within user specified limits. Tempest

leverages existing GNU compiler collection (gcc/gf77/g++) [18] to support transparent function-level thermal profiling, and performance profiles for any source code written in C, C++ or FORTRAN. This is accomplished by using built-in compiler instrumentation to annotate object code and linking to our tempest library. However, Tempest also exports compiler independent API for user aware profiling and regulation. At runtime, Tempest collects and parses the raw performance and thermal data and after a successful program termination, prints a summary to standard output. Results include time and temperature measurements for each function executed by the program.

Tempest does not significantly alter the performance of the scientific applications measured. Tempest incurs less than 7% overhead for temperature and performance profiling and less than 10% overhead for maintaining temperature within threshold values. Tempest is as portable and as easy to use as common Linux tools such as gprof [19] , allowing experts and non-experts alike to profile temperature and validate thermal reduction techniques.

There are several contributions in this work:

- *We propose, implement, and validate Tempest, a middle-weight framework for profiling sequential and parallel application thermals and validating thermal management techniques.*
- *We use Tempest to provide thermal profiles of several classes of parallel applications from common benchmarks including NAS PB.*
- *We demonstrated how Tempest can be used to profile and analyze the effects of thermal optimizations on a parallel application.*
- *We show how Tempest can regulate the thermals of an application and contribute to improving the overall reliability of the components in a system.*

This thesis is organized as follows. Chapter 2 describes related work to place our work in the context of ongoing research in thermal profiling and management techniques as well as work on system power and energy optimization. Next Chapter 3 presents the elements

of the Tempest framework. Chapter 4 illustrates the profiling results followed by an analysis of the thermals at function-level granularity through various sequential and parallel benchmarks. In this chapter we also show how Tempest can be used to quantify the effects of thermal optimizations such as dynamic voltage scaling. Chapter 5 contains concluding remarks with a discussion on Tempest and its limitations. Chapter 6 discusses briefly the ongoing and future work.

Chapter 2

Related work

This chapter describes related work to place our work in the context of ongoing research in thermal profiling and management techniques as well as work on system power and energy optimization.

To the best of our knowledge there have been no studies of the thermal properties of parallel scientific applications on real systems. There have been a limited number of thermal studies of commercial systems, particularly at the single system level. Traditionally thermal studies were relegated to the domain of mechanical engineering.

Simulators and Emulators

In 2003, Skadron et. al. proposed a simulation-based approach called HotSpot [20] and made the case for thermal considerations early in microprocessor layout design [20]. Unfortunately, ascertaining the effects of heat dissipation can be cumbersome. HotSpot and tools proposed by others [6, 21, 22] primarily use a combination of architecture simulation and thermodynamic models to quantify the heat dissipation of single system components.

Recently, researchers from Rutgers created a tool suite called Mercury [23] that provides system-level thermal profiling data using a model-based emulation approach. The heat

flow models are discretized to increase emulation speed and thermal sensors are used to validate emulation within 1 degree Celsius. The primary use for this toolset is to emulate thermal emergencies and test techniques to reduce heat damage, interruptions and migrate tasks etc.

Other studies e.g. Gurumurthi et. al. [24, 25] also included modeling temperature-aware disk scheduling policies and simulating thermal emergencies in data centers. This group from Penn state recently also studied effects of workloads using real disk I/O on thermals through simulation.

Recently, researchers from Duke studied temperature-aware workload placement policies [26] in the context of data centers to reduce cooling costs using modeling and simulators.

Real measurements

Bellosa et. al. [27, 28] propose the use of hardware counters to predict power and thermal profiles. The basic approach is to identify a correlation between event counts and power or thermal properties. Then, an analytical model is created using statistical regression to identify coefficients for a given application. The result is a model that predicts thermal temperatures based on performance data. Unlike simulation, such models are very fast but inflexible. For example, these techniques do not extend beyond certain components (e.g. CPU) due to the dependence on hardware counters.

There has also been some thermal-aware work related to data centers for non-interactive commercial workloads. For example, global scheduling across thousands of systems has been proposed to reduce the energy required to cool data center clusters [29, 30]. These studies propose algorithms that use real-time temperature monitoring data in the aisles between servers to optimally distribute work to the servers in cooler regions of the data center. This is shown to reduce the load on data center cooling equipment thus realizing significant operational cost savings. ConSil [31] surmises the temperature at inlets in data centers by using neural networks and real sensor information.

To the best of our knowledge, there are no approaches to create a real-time tool that provides a fine-grain thermal profile of an application at the function or at a block level. Parallel scientific applications are inherently interactive, phased-based and suffer dependencies across and within nodes. This implies fine grain profiling is necessary to identify and correlate thermal properties to source code. There are no simulators currently available that accurately model performance and thermals for clusters. The Mercury emulation tool is capable of modeling small clusters, but it requires a deep understanding of the heat flow in a system and is not easily ported. Our Tempest tool is orthogonal to these ongoing efforts and provides a tool that is accurate, portable, and collects details sufficient for analyzing parallel scientific codes.

To the best of our knowledge, this is the first work that presents measured results from thermal sensors for scientific applications correlated to source code. Also, we identified several other techniques that can be used to regulate CPU temperature. Further, we believe this is the first study of the use of dynamic voltage and frequency scaling as a runtime technique to satisfy user constraints on cluster-wide thermals while simultaneously maximizing performance.

Chapter 3

Tempest Framework

This chapter provides a detailed description of the elements in the Tempest runtime framework for parallel and sequential applications. This chapter is categorized into four sections: the components of Tempest aware applications, the developmental aspects of Tempest aware applications, our design evolution, and the details of current implementation. Finally, we conclude this chapter with a discussion that illustrates the manner in which the framework helps address the research challenges discussed in Chapter 1

The Tempest framework takes a runtime adaptive approach to profile and regulate the thermals of an application. In order to facilitate transparent thermal profiling and management, the framework takes a component-based approach. The Tempest framework provides shared object files (.so) as components to be linked with the application. One or more of these components must be linked to construct a complete Tempest-aware application. The framework addresses both thermal-aware and non-thermal-aware users while providing portability and transparency.

Figure 1 illustrates the components of the Tempest framework: profiler, controller, shared libraries, parser and stubs. Tempest operates completely in user-mode and does not require any modifications to the underlying operating system.

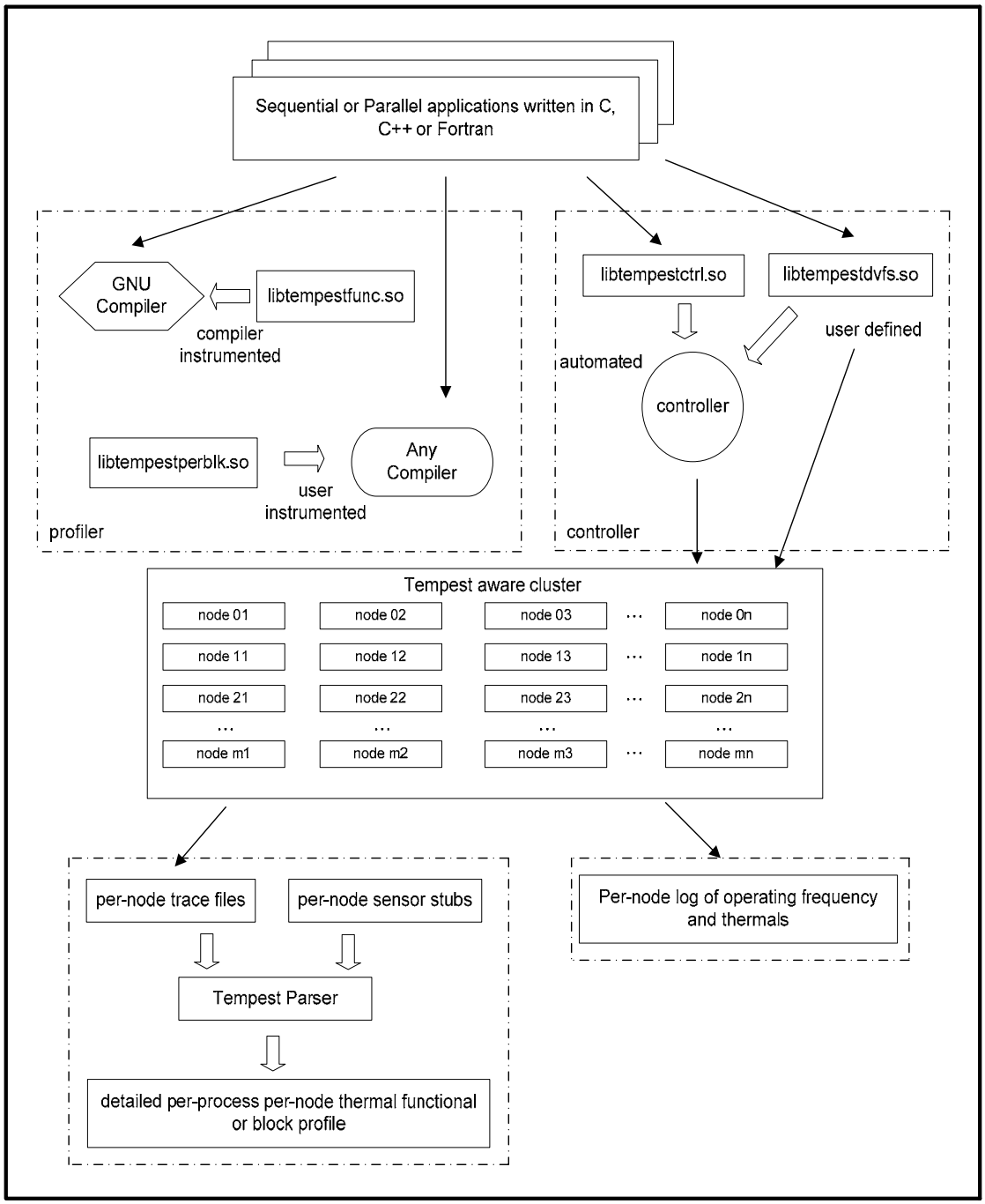

Figure 1: Components of Tempest framework: profiler, controller, shared libraries, parser and stubs.

Components of Tempest

The Tempest framework provides the following components for a Tempest-aware application.

Thermal Profilers

Tempest provides two profilers each with different levels of granularity and transparency. Thermal profiling can be achieved either at the granularity of a few lines of code (block level) or at the functional level. In both cases, the profiler measures real-time thermal data from individual nodes in a cluster using hardware sensors and correlates this data with program execution. Tempest requires an application to be linked with `libtempestperblk.so` and explicitly reference API in order to perform profiling at a block level.

While functional profiling is completely transparent to the user i.e. (without involving any explicit API calls), the programs must be linked with `libtempestfunc.so`. Further, while Tempest aware applications require source code recompilation and/or re-linking presently, runtime profiling and analysis is automatic and transparent.

Parsers

On successful termination of an application, Tempest generates several trace files that contain detailed timing information and critical thermal data obtained from hardware sensors. Tempest framework provides a parser to analyze and generate the thermal profile for scientific applications. The parser correlates the timing data and thermal information obtained during runtime and provides an estimate of the thermal characteristics of an application. This analysis is performed for every node in the cluster, since a parallel application running on many nodes generates multiple traces across a cluster. Tempest also generates a detailed profile of the application during post-runtime analysis.

Thermal Sensor Stubs

Tempest provides sensor stubs required by the parser. Each node has a sensor stub, which extracts specific thermal information from nodes in the cluster. Stubs are simple shell scripts that extract temperature readings from underlying hardware sensors. It is common to see that in a homogenous cluster (with identical node configurations) the thermal sensors can vary. Hence in order to facilitate ease of maintenance and robustness of Tempest parsers we isolated this extraction mechanism to sensor stubs making profilers (shared libraries) and parsers only architecture dependent and not vary within the same architecture. In short, these stubs are machine specific and may vary for nodes within the same cluster. The parser uses these stubs during post runtime analysis.

Controller

Tempest provides two shared libraries that a programmer can link with an application in order to regulate its thermals within a stipulated range. The static profile-driven controller requires a thermal-aware programmer to explicitly include API provided by `libtempestdvfs.so` in the application source code, while the adaptive runtime controller `libtempestctrl.so` regulates thermals transparently but requires code recompilation. Using a profile-driven controller it is possible to optimize for thermals and energy whereas the adaptive runtime controller is designed to primarily regulate the thermal behavior of an application and may not offer any energy gains.

Using one or more of these components it is possible to regulate the temperature of an application. Developing applications using Tempest is simple and straightforward. In the following sections of this chapter we discuss some of the API provided by the framework and issues in building Tempest-aware applications. The complete sources and detailed discussion on API is available [32].

Developmental aspects of Tempest aware applications

The Tempest framework offers its services in the form of shared libraries and binary images. It contains four shared-libraries that encompass the API. Table 1 briefly describes these shared libraries. Users can profile their code at a granularity of block or functional level. Figure 2 depicts the pseudo code for a generic application compiled with instrumented code and linked with `libtempestfunc.so`. The API represented in grey is

Tempest Libraries	Functionality	Usage	Issues
libtempestperblk.so	Provides API to profile applications at the granularity of few lines(block) of code or functions	Link with application sources	Require programmer support to call API
libtempestfunc.so	Provides function handlers to intercept function calls and profiles the applications at a functional level	Compile application sources with *-finstrument-functions* option using GNU Compiler (gcc/f77/g++) collection before linking	Transparent
libtempestdvfs.so	Provides API to control the CPU clock frequency and regulate the thermals and power consumption	Link with application sources	Require programmer support to call API
libtempestctrl.so	Provides API to control the thermals of an application at runtime	Link with application sources	Transparent

Table 1: lists the shared libraries provided by the Tempest Framework. Functionality, usage and issues associated with each library. API details are mentioned under

transparent to the user. Every function call made by the application is intercepted at 1 and 2 upon entry and exit respectively. These API are exported by the GNU compiler collection and implemented by Tempest to facilitate function-level instrumentation.

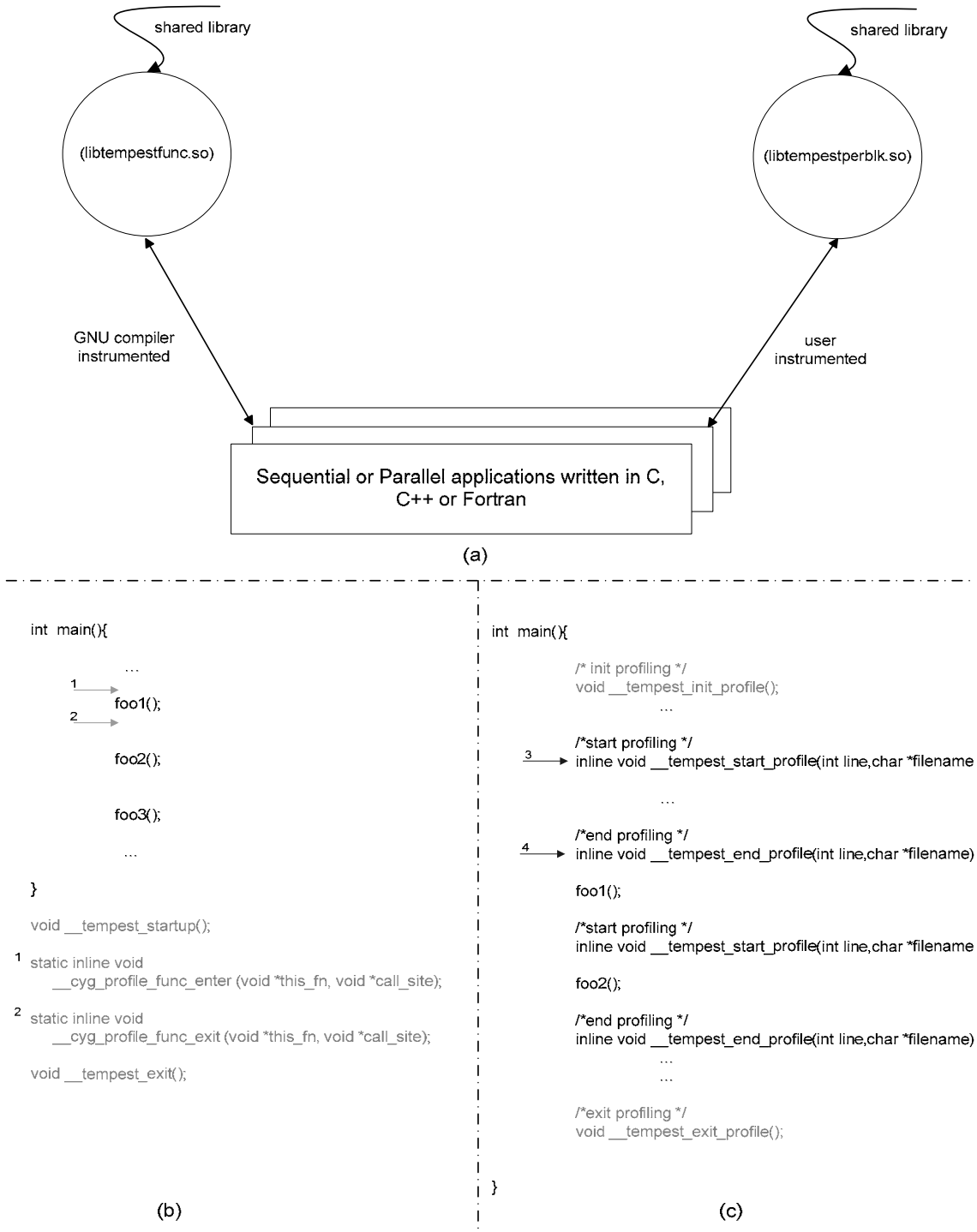

Figure 2: (a) A generic Tempest aware application with thermal profiling. (b) and (c) Linking pseudo code for function and block level profiling respectively. The API in listed in grey are transparent to user.

In order to profile a chunk of code the API (3 and 4) must be explicitly referenced just before and immediately after the block of code respectively. Figure 2 (c) depicts pseudo code for block level profiling achieved using `libtempestperblk.so`. These API are compiler independent.

Tempest also provides API to regulate the thermals of an application. Figure 3 shows pseudo code for controlling temperature at runtime using a heuristic history based PID controller. The controller (`libtempestctrol.so`) regulates the temperature by adaptively changing the CPU frequency and voltage. The controller includes the DVFS API shown in Figure 3 (c) and implemented in `libtempestdvfs.so`. The API (3 and 4) along with many others that Tempest DVFS library supports can be explicitly referenced by the programmer.

Currently, developing tempest-aware applications entails source recompilation and linking with shared libraries. Tempest also supports simple configuration files to facilitate profiling preferences. From a usability perspective, developing tempest-aware applications is simple and straightforward.

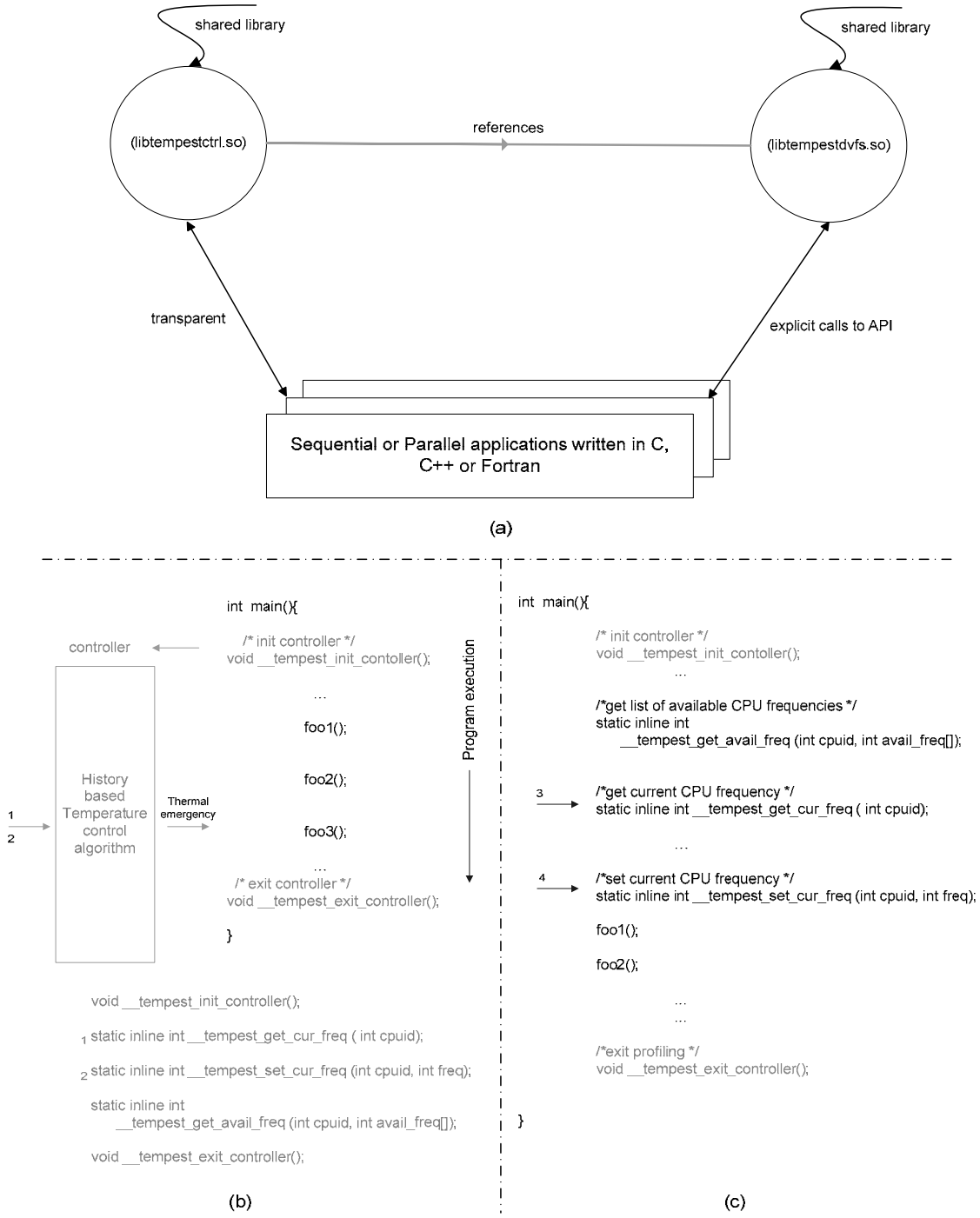

Figure 3: (a) A generic Tempest aware application under thermal control.
(b) pseudo code for adaptive runtime thermal regulation (c) pseudo code for static profile-driven thermal regulation. The API's listed in grey are transparent to user.

Design Evolution

We had several design goals for Tempest. First, since we had experience using simulation-based thermal measurement tools like HotSpot [20], we wanted to create something that was significantly easier for non-experts to use. We attempted to minimize complexity for both use of the tool and understanding of the results. Second, we wanted to maintain portability. Our initial application of the tool was thermal profiling of parallel applications on various systems. This work required a tool that was easy to port. Third, we wanted the tool to be as non-intrusive as possible. This was a significant challenge since sampling thermal sensors can have high overhead and we required fine-grain measurements to correlate temperature to source code.

We iterated through several designs before converging on our current implementation of the Tempest framework. We initially set out to create a tool similar in design and functionality to gprof, a popular performance profiling tool. Unfortunately, we had to abandon this idea early on since gprof tracks source code to execution mappings at a different granularity. For example, gprof creates equal sized buckets for the entire process address space and samples the program counter around 100 times a second and then maps the address ranges back to application sources before listing the time spent in various functions; gprof does not pinpoint which function was executing at a particular instance of time in a program. Nonetheless, Tempest requires a function-level or block-level timeline since temperature readings from sensors occur and vary in real time throughout the duration of a code. It is quite possible that the same function may execute at different temperatures during an execution as conditions change with real time. Hence, simple modifications to gprof would not be appropriate since it did not offer the level of control, detail, and accuracy we required for thermal profiling.

Due to our portability goal, we avoided integrating temperature measurement and process monitoring within the OS kernel, even though it could result in lower overhead and would be completely transparent to the user. For similar reasons we decided against

binary rewriting of the running application and modification of GNU glibc [33]. We plan to reconsider both of these approaches in succeeding versions of the tool. Consequently, we adopted two approaches, one that relies on the compiler to provide instrumentation hooks for tracing the execution of the program for function-level profiling and another approach which involved no compiler support but required the programmer to explicitly call the profiling API. In either case we provide a user-level library that is easily portable and can be linked to the application at compile-time to generate run-time thermal measurements.

Implementation and Discussion

The Tempest Framework is currently implemented on GNU/Linux. Some of the components of the implementation are architecture dependant. Therefore, in order to follow the discussion in subsequent sections, familiarity with x86 and Power PC instruction set, systems programming, shared libraries and knowledge of configuring hardware sensors and its related issues is required. This thesis provides a description of the implementation of the core aspects of the framework which are the profilers and controllers. The basic implementations of parsers, thermal sensor stubs are fairly observable and are not elaborately discussed in this work.

Profiler

The profiler (per-block or functional) provides a mapping between the program execution and its thermal characteristics. We target scientific parallel applications that are running on large scale clusters with dedicated number of processors where only a single application runs on one or more nodes. This assumption is valid because job schedulers in clusters run a job on either fixed or variable number of nodes but do not share the same nodes between two applications at the same time. In order to profile the application for

thermals, our approach involved: 1) measuring the entry and exit times of a function or block of code; and 2) measuring the temperature during the course of a function's or block's execution. Figure 4 details how the parser interprets the mappings just described.

Function-level profiling is implemented as a shared library that leverages support from the GNU compiler. The GNU compiler exports function-handlers to applications compiled with `-finstrument-functions` option. By implementing these handlers we were able to determine the function entry and exit instances. However, in order to measure entry and exit times, we needed a lightweight timer which could give us a CPU time stamp. We avoided using timer functions provided by the system as they are known to incur significant overhead, changing the true nature of the application's execution. Instead we opted to use a time stamp counter to measure the CPU time stamps using the `rdtsc`[2] instruction. This approach is totally transparent to the application programmer. However block-level profiling requires programmer support in the form of explicit library calls from application sources.

In order to measure temperature while the program is running, we created a light weight ***temp****erature measuring* ***d****aemon* (*tempd*). This *tempd* periodically (four times a second) measures temperature through hardware sensors on the motherboard and the processor using Linux hardware monitoring lm_sensors [2]. This sampling rate and selection of functions can be varied through a configuration file. The *tempd* is a part of the shared library and gets created before the actual `main` function of the application is invoked. The profiling information on every node in the cluster along with the timestamps is collected into a trace file. Upon executing the application and just before exit, the destructor in the shared library is called which sends a signal to *tempd* for termination and performs cleanup operations.

---

[2] Technically speaking, `rdtsc` is an x86 instruction. However, we identified the equivalent instruction set on the PowerPC architecture, further we are aware of the tradeoffs involved in using `rdtsc`. Our prime reason for this implementation choice is performance.

In this implementation we assume that the underlying hardware sensors are accurate and do a good job in reporting temperature. We validated the hardware thermal sensors for accuracy by running a set of CPU intensive micro-benchmarks. Finally the tempest parser acquires function timestamps and provides a mapping between timestamps and temperature for the workload on the cluster. Figure 4 depicts mapping of *tempd* to application sources. For example, total runtime of function with address 0x80e7c0 is obtained by measuring time stamp counter (tsc) values p_2 and p_3 just before entering and immediately before exiting the function respectively. The total execution time of code observed in seconds can be calculated using Equation [1] below

Total execution time $= (p_3 - p_2) / CPU\_CLK\_FREQ$ Equation [1]

This execution time includes the runtime of `foo` and all its children if they exist. The thermal characteristics of `foo` are obtained by analyzing the temperature readings closest to p_3 and p_2 in *tempd* i.e. t_6 and t_{11} respectively. The parser then reads the `symbol-table` of the executable to map addresses of functions to their names to generate a human-readable functional temperature profile. This mapping provides a complete thermal profile of the application. However in the case of block-level profiling, the tsc values corresponding to line numbers and source filename are used to generate the thermal profile.

In order for the time stamp counter values to make sense, the application should always run on the same core throughout its lifetime as a result the profiling library transparently binds the Tempest process and its corresponding *tempd* to the same fixed CPU core. This is essential in the case of CMP and SMP machines because the tsc values may vary between cores and processors. Consequently, the process is ensured to read the same time stamp counter even after a context switch throughout its lifetime. Further, it is required that for Equation [1 to hold true, the CPU frequency should not be altered while profiling. However, it is possible to accommodate profiling applications that run with variable CPU frequencies but this version of the Tempest framework mandates a unique fixed frequency.

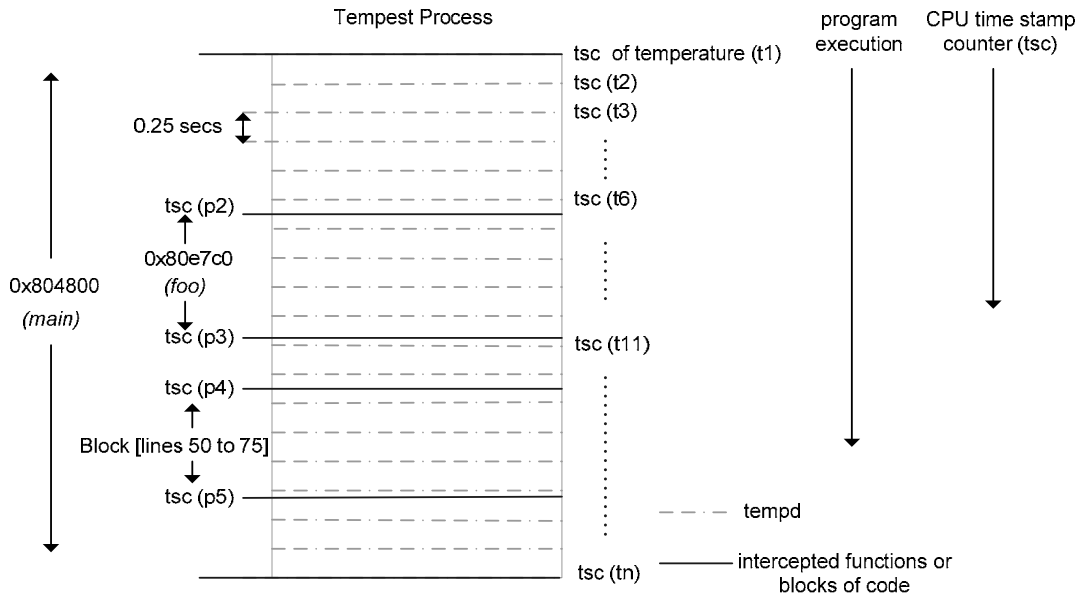

Figure 4: A Tempest process. The parser approximates CPU time stamp counter (tsc) values of function calls or code blocks in the actual application to tsc values obtained from tempd. This facilitates in attributing run-time thermal values to application sources.

Controller

The static profile-driven and adaptive runtime Tempest controllers regulate the CPU temperature during program execution. We continue our discussion with the assumption that only a single application is responsible for contributing to the system thermals. In order to regulate CPU thermals, our approach involved: 1) designing an algorithm that efficiently controls temperature with minimal performance impact and also saves energy whenever possible. 2) identifying the best frequency possible while taking into consideration the thermals. Figure 3 depicts the transparent and programmer specified API to regulate the temperature. The controllers do not profile the application, i.e. they do not provide any thermal information to the user like the profilers.

Adaptive Runtime PID Controller

Figure 1 illustrates the role of our controller in the Tempest framework. The adaptive runtime Tempest controller regulates CPU temperature during program execution. To regulate CPU thermals, we designed systems-software that efficiently controls temperature with minimal impact on performance. Our controller monitors and reacts to systemic information making it generally applicable to any system and application though we focus on parallel scientific applications. In our analyses, we use Tempest to correlate thermals to source code and thermals to CPU clock frequency.

The adaptive runtime PID controller does not require any source code modifications except for linking with this library. As mentioned earlier, we use lm_sensors[2] to read system-wide thermals. However, in order to control the CPU temperature we experimented with several approaches consisting of Dynamic Voltage and Frequency Scaling (DVFS), adaptive CPU fan speed control, calling `nop` instructions and finally using `usleep` system call to enable a process to preemptively relinquish the CPU. Using the latter two approaches it is possible to cool the CPU but the performance suffers since they halt the program execution. Thus, we have considered the resulting performance loss unacceptable to HPC applications. Further, adaptively controlling the CPU fan speed is not trivial and is not supported in many systems. The present interface surrounding regulation of fan speed is hardware specific and varies greatly with systems. Hence for reasons pertaining to portability we did not support this approach in our present implementation.

Thus, our approach centers on the use of DVFS to reduce temperature. Initially we studied use of both static and heuristic temperature regulation techniques. Static controllers, where power mode schedules are determined at compile time, do not adaptively adjust to any changing requirements during program runtime. Heuristic approaches can often react to runtime requirements using history based predictors. We developed a heuristic history based PID controller to modulate CPU temperature. The

controller predicts future CPU temperature based on current and previous history of temperatures.

We track the temperature change by monitoring the CPU sensors discretely at fixed intervals of time. A feedback controller adaptively alters or maintains processor frequency based on the error in temperature or the difference in current temperature and threshold temperature. This error is a prediction of how much hotter (or cooler) the CPU will be in the next time interval.

PID Controller

In this section we briefly describe our PID controller. Let T_m be a minimum temperature threshold and also the temperature we ideally want the CPU to sustain. We formally define error at time t as $e(t) = (T_m - T_t)$, or the difference between the ideal temperature and the temperature at time t.

If $e(t) \geq 0$, then we are under thermal limits and if the CPU is in a low power state, we may want to increase CPU frequency to ensure that high-performance is maintained. Otherwise, the current temperature is greater than the threshold temperature (T_m); thus we should reduce CPU power and frequency to meet the thermal constraints assigned by the user. A PID controller adjusts its output (in our case a CPU voltage and frequency setting) based on three terms: a term proportional to the current error, a term proportional to the integral of the error, and a term proportional to the deviation of the error. We can thus express a PID controller as:

$$T(t+1) = k_P e(t) + k_I \int e(t)dt + k_D \frac{de(t)}{dt}$$ Equation [2]

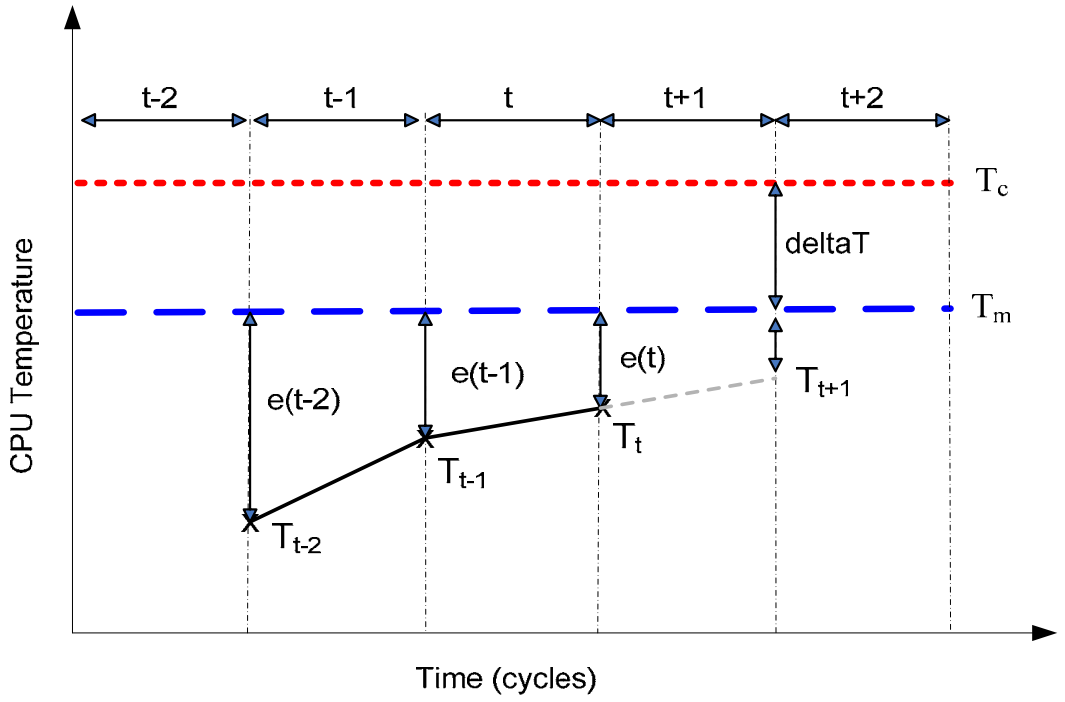

Figure 5: Tempest PID controller uses past temperature information to predict the next temperature phase. Users specify a critical threshold. We use the sampling rate to determine a minimum threshold (or setpoint), a buffer to reduce the likelihood of exceeding the user-specified temperature. If the predicted temperature exceeds the minimum threshold, the controller reduces the processor voltage and frequency.

where $e(t)$ is the error at time t and k_P, k_I and k_D are coefficients of the proportional terms. The term proportional to the current error is denoted by $k_P(T_m - T_t)$. This term is an indication of the current status of temperature. If this value is comfortably below the threshold value, then the controller switches the CPU to higher frequencies by actuating the API provided by `libtempestdvfs.so`.

The history of previous errors in the last two cycles can be estimated by calculating the integral of errors for the history as $\int_{i=t-2}^{i=t}(T_m - T_i)dt$ which can alternatively be stated as the

term proportional to the integral error. This is formally represented as $k_I \int_{i=t-2}^{i=t} (T_m - T_i)dt$. This term illustrates the net error of all the intervals including the present interval. However this does not indicate a trend in the errors.

In order to observe trends in rate of increase or decrease of errors we approximate the temperature gradient by considering the last and first points, which can therefore be stated as the rate of change in errors with time. Thus, this rate of change of errors or the gradient can be approximated by calculating the differential of errors observed at T_t and T_{t-2}. This differential can be denoted by $k_D(e(t) - e(t-2))dt$. Therefore, the predicted error in the next interval can be shown as:

$$T(t+1) = k_P(T_m - T_t) + k_I \int_{i=t-2}^{i=t} (T_m - T_i)dt + k_D(e(t) - e(t-2))dt \quad \text{…Equation [3]}$$

The projected temperature is shown in Figure 5 in grey dotted lines in cycle (t+1). The coefficients k_P, k_I and k_D can be estimated empirically by iteratively running a set of microbenchmarks. However, this process can be cumbersome. As a result, in this thesis we circumvent this problem by considering the sign($\pm$) of the terms instead of their absolute values. This intuitive solution has certain tradeoffs. The limitation is that, we do not have an absolute value of $T(t+1)$. However, thinking retrospectively all we need is its spatial locality with reference to the threshold temperature (T_m) i.e. whether the future temperature lies below or above T_m. If the sign of all the terms is positive, then it can be inferred that $T(t+1)$ is under the threshold limits and similarly if the sign is negative, we can expect $T(t+1)$ to be above T_m. Based on this expected temperature and the current frequency the PID controller regulates the CPU frequency for the next time interval as shown in Figure 6.

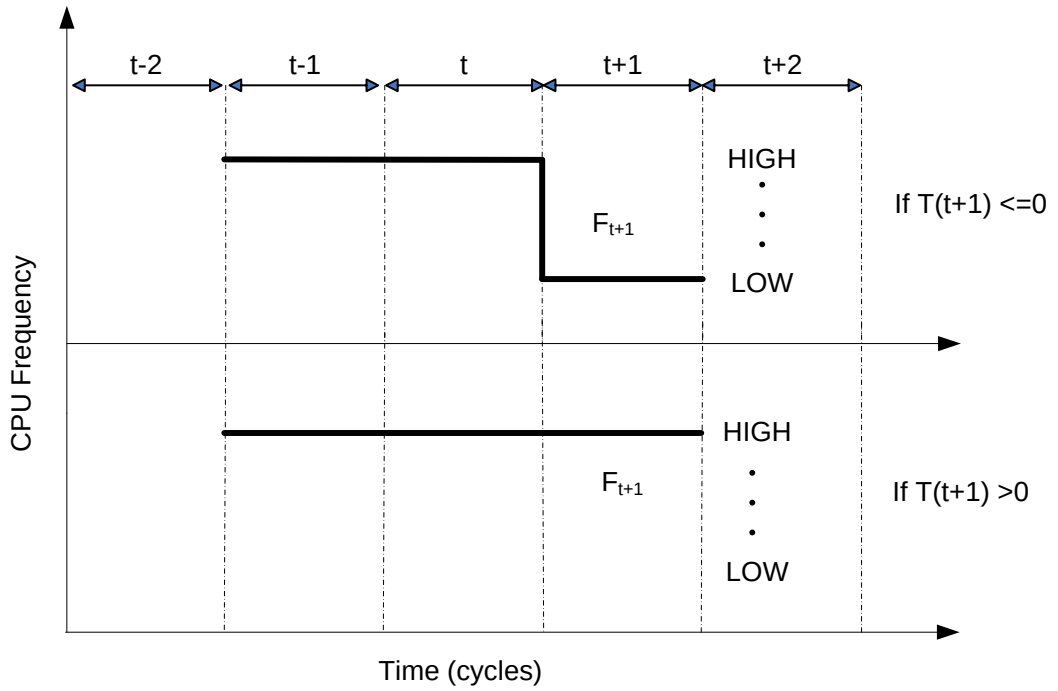

Figure 6: The runtime adaptation of processor voltage and frequency given the prediction of the PID controller.

Further, at the end of each cycle, the controller updates the three temperature points (T_{t-2}, T_{t-1} and T_t). The controller estimates T_{t+1} while T_{t-2} is updated to T_{t-1} and T_{t-1} is updated to T_t. The next section explains how the CPU frequencies are chosen.

Choosing the right frequency set

Normally processors support a wide range of operating frequencies, for example AMD 64 4000+ supports 2.4, 2.2, 2.0, 1.8 and 1.0 GHz. Each such frequency reacts differently to workloads and contributes to a different rate of increase in temperature. For CPU intensive workloads, operating at lower voltages can greatly reduce the heating effects caused by the application but may also reduce its performance. Our preliminary experiments concluded that tradeoffs exist between corresponding frequencies and their respective thermal responses. For parallel HPC applications, it is imperative to choose

appropriate frequency and manage thermals without significant impact on performance. This selection can be either estimated from the CPU data sheets or by running CPU intensive micro benchmarks and visualizing the thermal profile of a workload run at each frequency. For AMD 64, we found there was no significant drop in temperature as a result of scaling from 2.4 to 2.2 GHz and 2.2 GHz to 2.0GHz but there was a significant decrease in temperature between 2.4 and 2.0 GHz, and 2.0 to 1.8 GHz. As a result we choose to use frequencies that offered significant temperature difference. Further, in order to sustain the drop in temperature (whenever the frequency is reduced), the CPU is maintained in the low frequency for the next few subsequent cycles. However, significant thermal differences can be achieved when operating with other frequencies lower than 1.8GHz. Further, some of our test applications suffered a measurable negative impact on performance while offering considerable reduction in operating temperature at lower frequencies. As a result, the Tempest controller considers such frequencies in this implementation. Furthermore, while this version of Tempest does not automate the process of choosing the best frequencies we plan to automate this in future versions of Tempest framework.

Choosing the sample interval

If this sampling interval is too small, then we poll the hardware sensors frequently, incurring overheads and affecting the performance of the application. On the other hand, if this interval is too large, we may miss opportunities to reduce thermals or maintain performance since prediction will probably not be accurate. Also, the interval should be chosen to handle thermal spikes as well as other trends. We used a number of microbenchmarks that allow us to determine a good compromise sampling rate of every one second for our experiments. On the systems under test this sampling rate resulted in minimal performance impact and reasonable accuracy.

Choosing the threshold gap ‘deltaT’

Figure 5 illustrates a noticeable gap between the critical threshold (set by the user) and the minimum threshold (set by our system). Since the PID controller uses past history for prediction and the sampling interval is not infinitesimally small, misprediction may allow the temperature to exceed the PID set point. Thus using the critical threshold as the set point, we are likely to exceed the threshold often. Since we want this threshold to be a ceiling on the temperature as much as possible, we create a temperature buffer by using a lower temperature (the minimal threshold) as the setpoint for the controller. The controller tracks the buffered setpoint and is less likely to exceed the user-specified critical threshold.

We call this Threshold Gap *deltaT*. From the perspective of the sampling interval, *deltaT* is the maximum increase in temperature that can be observed over one sample interval. Also, *deltaT* should be large enough to allow the CPU to cool before reaching the user-specified critical temperature. In theory, this value can be deduced from the processor data sheets. We experimented with a range of values and we chose a value of 5°F.

Static profile-driven controller

Both the controllers use `libtempestdvfs.so` to perform dynamic voltage and frequency scaling (DVFS) of CPU. The static profile-driven controller exports the DVFS API to the user while the adaptive runtime controller uses these API transparently through a history-based algorithm. The implementation of the DVFS API is based on the cpufreq [34] kernel module and `/sys` interface. Figure 3 illustrates the API provided by this library. These API can also be explicitly called by the programmer from application sources. In order to have an insight as to where these calls are to be incorporated within the program sources, it is suggested that the programmer profiles the code first using Tempest profiler and then basing on the thermal profile, identify the thermal emergencies and then call the API as required.

Further, these API may also be used to save the overall energy consumed, without any performance hindrance for some applications. For example, in certain parallel MPI applications, it is not required to run the CPU at its peak voltage during the communication phase. Consequently, in applications with large phases of such communication, the programmer may make use of the DVFS API to minimize the energy without compromising performance.

Portability

Currently, the Tempest framework supports five architectures: x86, x86_64, CMP, SMP and Power PC. However, since most of the Tempest framework is fairly architecture-neutral, the implementation can be easily extended to other architectures like Sparc, Digital Alpha, and Cell etc. Porting to different operating systems like Windows and OS-X is also easily achievable. Since, most operating systems support hardware sensors, allow means to bind a process to CPU, and provide fine grained timing information it is easy to port Tempest across a wide range of operating systems and architectures.

Scalability

Tempest scales well with number of nodes in a cluster. In fact, each tempest-aware process is bound to a core or a processor in a node. The profiling and controlling is done on a per-node basis. Hence, the number of nodes in a cluster does not impinge the performance of Tempest libraries.

Summary

In this chapter we provided an insight into the elements in the Tempest framework for parallel applications. We also illustrated the manner in which the framework addresses the research challenges described in Chapter 1. The results are elaborately discussed in Chapter 4. Tempest framework empowers the user with the ability to study and analyze the thermal properties of applications. Through shared libraries, the framework supports both transparent and programmer-defined API to profile and manipulate applications at runtime.

All components of Tempest applications operate in user mode and do not require modifications to the operating system. This addresses the research challenge of maintaining portability. Tempest does real time thermal profiling and regulation running on native hardware and software. Further, using Tempest does not involve any simulation or emulation. This addresses our second research challenge, running on real machines (actual runs). Tempest does not mandate any restrictions on the number of nodes in a cluster, consequently scaling well with large scale clusters, thereby addressing our goal of scalability. Tempest is easy to use and caters to the needs of thermal-aware and non-thermal aware users, involving minimal or no code modifications. This addresses the goal of transparency. The framework also provides a meta-language for specifying profile configurations and output formats in a configuration file. Tempest provides previously unavailable insight into the thermal characteristics of applications running on real systems. In a broad sense, we found Tempest to be portable and to provide accurate, repeatable measurements.

Chapter 4

Case Studies and Results

This chapter demonstrates the applicability of the Tempest framework in the areas of high performance thermal-aware computing. Tempest provides previously unavailable insight into the thermal characteristics of applications running on real systems. In this chapter we present select results from various system and code implementations. We also discuss our automated system-level DVFS controller for thermal regulation. Tempest provides two shared libraries that an application should link to in order to regulate its thermals within a stipulated range. Thus, static profiling and control is possible by explicitly including API calls provided by Tempest in the application source code. Since this would require code recompilation, we propose a runtime, profile-driven controller designed to regulate the thermal behavior of an application while minimizing performance loss.

Though we could report all sensors for all systems, we found the ambient sensors located in various places in the system chassis are valuable but do not correlate significantly to source code phases and are more a reflection of external temperatures and airflow. Hence, we focused on the results as reported by the core CPU sensors and found that the thermals of an application have some basic trends that reflect the phases of an application. On the other hand, an interesting observation was that the thermals vary between significantly systems (under the same load).

Experimental Corroboration

We tested Tempest on a number of systems and compilers first using a series of micro-benchmarks, toy codes SPEC (cpu2000), HPL, NAS-OMP, NAS-Serial, and NAS-MPI. Tempest measures data from all available thermal sensors. On the systems we measured, we observed as few as three sensors on some x86 platforms from AMD and up to seven sensors or more on PowerPC G5 systems. Tempest will run successfully on any Linux-based system that has onboard sensors and supports the lm_sensors, a Linux tool that provides system-level access to hardware sensors. As mentioned, Tempest currently supports `gcc` for C code, `g++` for C++ code, and `g77` for FORTRAN code for function-level profiling. However for block-level profiling, Tempest support is independent of compilers.

We verified the correctness of Tempest against the aforementioned gprof tool. gprof provides an estimate of the time spent in each function of the program. We performed experiments where we ran the vanilla code, using the gprof tool, and then using the Tempest. gprof introduced less than 10% overhead over the original code for all codes measured including the SPEC CPU 2000 benchmarks and the NAS Parallel Benchmark suite. Tempest showed less than 7% overhead for the same codes. Repeated measurements were subject to variance of about 5%. Both tools provided similar results for total execution time in the various code functions within the variance mentioned.

Experimental setup

Thermal sensor technology is emergent and at times unstable, so we attempted to run Tempest on as many systems as we had access to in an effort to illustrate portability and usefulness. Systems also had to support lm_sensors or its equivalent software. Systems include four-node dual-core and dual-processor AMD 1.8GHz Opteron system running a 2.6.9-11 Linux Kernel, the System X supercomputer (PowerPC G5), four node heterogeneous cluster with AMD 64 3200+, 4000+ and P4 processors and an 8-way SMP

Opteron system. We also tested Tempest with several x86 and x86_64 machines with both shared and distributed memory. On all systems, we used GNU C, C++, and/or FORTRAN compilers.

For all experiments pertaining to the profiler (except those noted later) we disabled DVFS and auto fan speed regulation to circumvent all thermal feedback effects. This effectively sets the CPU to its highest frequency and sets the fan speed to a constant high speed (e.g. 3000 RPMs). We ensured that the cluster was running bare minimal services in order to eliminate any thermal noise caused by unnecessary daemons. In order to detect potential feedback effects, we measured the steady-state system temperature by running the *tempd* process without any workloads. We observed that *tempd* had no impact on the system temperature, and in fact used less than 1% of CPU time. We allowed the system return to a steady state (ambient or system room) temperature after every test. We repeated our experiments multiple times and with multiple configurations to check for consistency.

A:	B:	C:	D:	E:

`main()`
`{`
 	
`}`
	`main()`
	`{`
	`foo();`
	
	`}`
		`main()`
		`{`
		`foo1();`
		`foo2();`
		
		`}`
			`main()`
			`{`
			`foo1();{`
			`foo2();`
			`}`
			`foo2();`
			`}`
				`main()`
				`{`
				`foo1();{`
				`foo1();`
				`foo2();`
				`}`
				`foo3();`
				`}`

Table 2: Lists some of the Micro-benchmarks that test Tempest's correctness under various interleaving and recursion conditions. These benchmarks test the following: A (main alone), B (one function), C (multiple functions), D (multiple functions with interleaving), and type E (multiple functions with recursion and interleaving)

In the following section, we present profiler results for some of the serial and parallel benchmarks.

Using Tempest Profiler for Serial Micro-benchmarks

In our first set of experiments, we developed micro-benchmarks to test the Tempest tool under extreme conditions. Table 2 shows our micro-benchmarks for various `foo` functions. We were ensuring that the sensor data was being traced correctly and that the thermal profiles were as expected. Table 3 shows the results in standard output format and Figure 7 illustrate the thermal profile output for micro-benchmark D where the `foo1` function dominates total execution time running a CPU burn benchmark while `foo2` simply exists after a short timer expires.

Table 3 shows Tempest output divided horizontally into functions (`main, foo1, foo2`) ranked by total execution time spent in each function. The total time heading for each function provides the amount of time spent in that particular function. Since main calls all functions in this case, it's total time is the duration of the program. However, `foo1` accounts for most of the time spent in `main`. `foo2` accounts for less than 1 second of the total time. Since the time spent in `foo2` is small relative to the sampling interval for the thermal sensors, thermal statistical data is not reported in some cases (e.g. Avg). For `foo1` and `main`, all thermal data is shown for each of the two thermal sensors. The `foo1` function is designed to heat up the CPU, which it does fairly quickly as shown by the Avg and Max temperatures.

Figure 7 plots the temperature trends for each function. Note the y-axis is temperature in degrees Fahrenheit and the x-axis is total execution time in seconds. Also, the duration of each function is show across the top of the figure. As described previously, `foo1` steadily increases the temperature of the CPU until `foo2` is finally called at which point the temperature drops abruptly. Recall that processor and fan speed are fixed for the duration of these experiments. Thus, we are limiting the thermal effects to those of the application.

Function: `main` Total Time(sec): 59.860001

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	114.000	120.716	124.000	2.729	7.450	121.000	114.000
sensor2	94.000	95.117	97.000	0.565	0.319	95.000	95.000

Function: `foo1` Total Time(sec): 59.828545

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	114.000	120.716	124.000	2.729	7.450	121.000	114.000
sensor2	94.000	95.117	97.000	0.565	0.319	95.000	95.000

Function: `foo2` Total Time(sec): 0.000000

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	114.000	nan	114.000	nan	nan	114.000	0.000
sensor2	94.000	nan	94.000	nan	nan	94.000	0.000

Table 3(above): Tempest functional profiling results for micro-benchmark D.

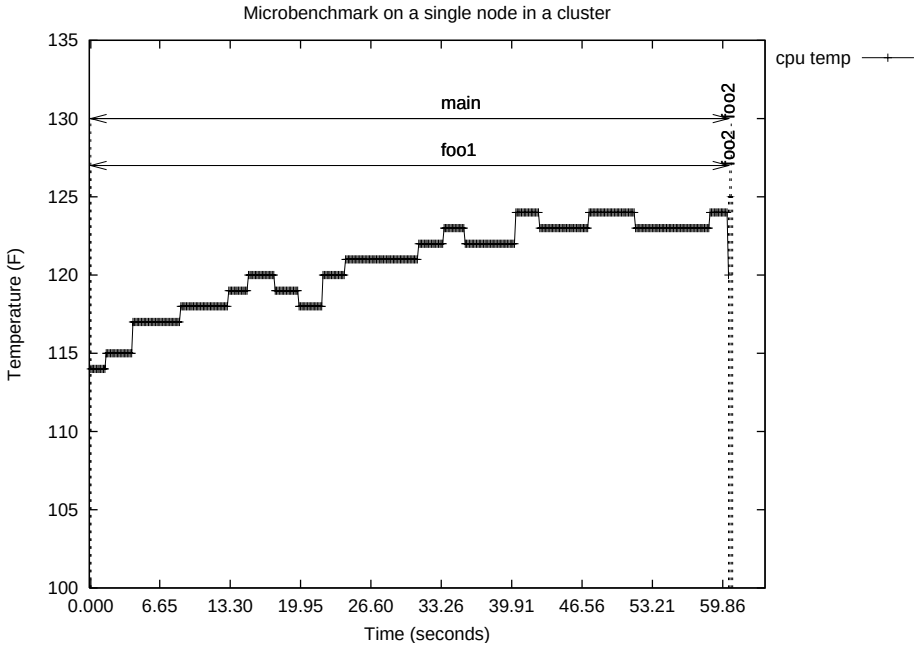

Figure 7: illustrates that temperature increase is dominated by the `foo1` function calling a CPU burn code that heats the CPU rapidly.

Using Tempest Profiler for Parallel Benchmarks

This section provides a description of Tempest-aware parallel scientific benchmarks. In our second set of experiments, we used common application suites from the parallel computing community to benchmark cluster systems. For brevity, we focus on codes from the NAS parallel benchmark suite. Figures 8 through 15 illustrate the temperature profile data for eight of the NAS PB codes. Each graph for a single code is a series of vertically stacked axes with y-axis for temperature in Fahrenheit and x-axis for time in seconds. Each vertical graph for a single code corresponds to node 1, node 2, node 3, and node 4 respectively. The vertical graphs for a single node are vertically aligned so as to aid identification of phase trends in the application. Tables 4 through 11 show the standard output data from Tempest for the same runs on one of the nodes. We now use the observed results to classify thermal profiles of these parallel applications.

The BT benchmark performs several tasks followed by a synchronization event that occurs at about 1.5 seconds into the run for our class W experiments depicted in Figure 8 and Table 4. These nodes exhibit very interesting behavior. This is one of the few codes in the suite that has obviously synchronized thermal characteristics with time. At the synchronization event, all nodes see a dramatic rise in temperature indicative of increased computation. Though the trends are similar across nodes, some nodes run hotter than others which was interesting. For instance nodes 1 and 4 jump above 105 degrees while node 2 stays below and node 3 runs very hot comparably at over 110 degrees.

The CG benchmark is is a conjugate gradient solver and is computationally intensive. The thermal profiles are not as clearly defined at synchronization points as they are for BT, however there is a clear warming trend that appears to be synchronized across all the nodes. Nodes 3 and 4 are running hot this time crossing 115 degrees quite rapidly while nodes 1 and 2 are somewhat cooler never actually breaching 115 degrees. Though the code behavior is similar across nodes, there is no discernible thermal behavior among any two nodes. Thus the local workloads appear to have significant effects on the local

temperatures.

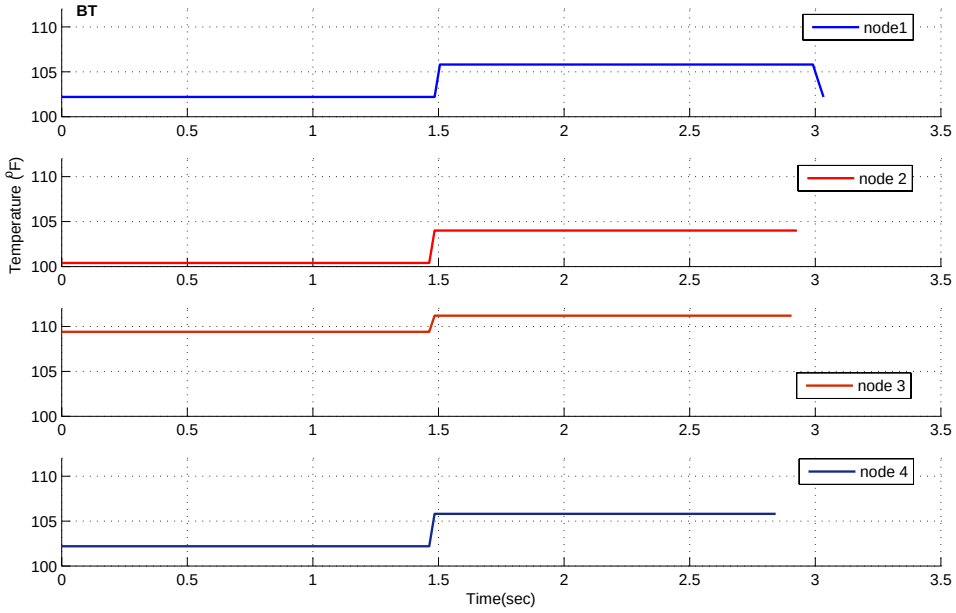

Figure 8: Thermal profile of BT benchmark, NP=4 and Class = W.

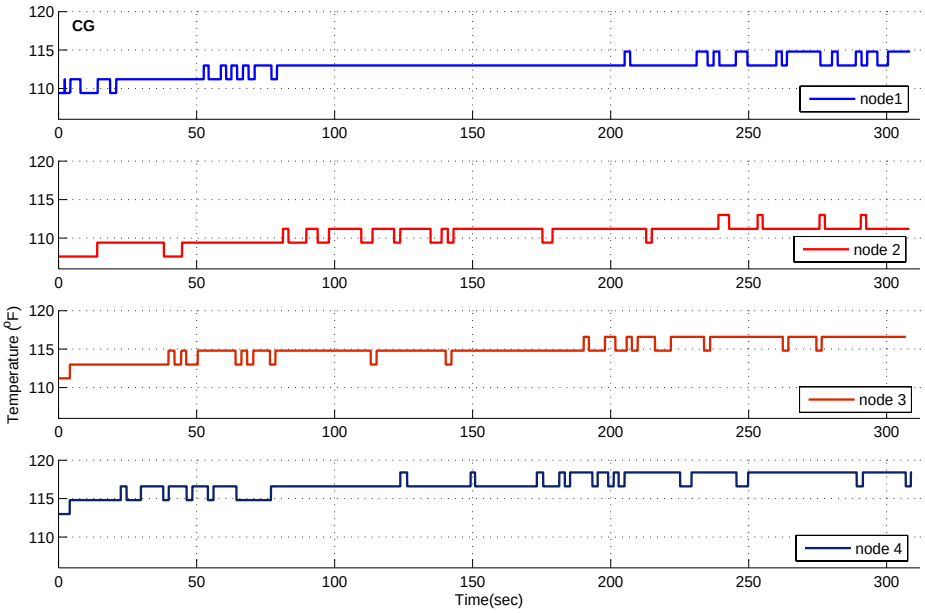

Figure 9: Thermal profile of CG benchmark, NP=4 and Class=C

Function: adi_ Total Time(sec): 6.320194

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.000	91.000	0.00	0.00	91.000	91.00
sensor2	93.20	93.200	93.200	0.00	0.00	93.200	93.20
sensor3	104.00	104.00	104.00	0.00	0.00	104.00	104.00
sensor4	102.20	103.96	105.80	1.80	3.24	102.20	102.20
sensor5	113.00	113.01	114.80	0.16	0.02	113.00	113.00
sensor6	102.20	102.20	102.20	0.00	0.00	102.20	102.20

Function: matvec_sub__ Total Time(sec): 4.081683

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.00	91.00	0.00	0.00	91.00	91.00
sensor2	93.20	93.22	93.20	0.02	0.00	93.20	93.20
sensor3	104.00	104.00	104.00	0.00	0.00	104.00	104.00
sensor4	102.20	103.98	105.80	1.80	3.24	102.20	105.80
sensor5	113.00	113.04	114.80	0.16	0.02	113.00	114.80
sensor6	102.20	102.22	102.20	0.02	0.00	102.20	102.20

Function: matmul_sub__ Total Time(sec): 3.797554

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.00	91.00	0.00	0.00	91.00	91.00
sensor2	93.20	93.22	93.20	0.02	0.00	93.20	93.20
sensor3	104.00	104.00	104.00	0.00	0.00	104.00	104.00
sensor4	102.20	103.97	105.80	1.80	3.24	102.20	105.80
sensor5	113.00	113.01	114.80	0.16	0.02	113.00	114.80
sensor6	102.20	102.22	102.20	0.02	0.00	102.20	102.20

Table 4: Partial Tempest functional profile for BT benchmark with NP=4, class W.

Function: vecset_ Total Time(sec): 2.151604

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	93.00	93.00	93.00	0.00	0.00	93.00	93.00
sensor2	92.30	92.30	92.30	0.00	0.00	92.30	92.30
sensor3	104.00	104.00	104.00	0.00	0.00	104.00	104.00
sensor4	105.80	105.80	105.80	0.00	0.00	105.80	105.80
sensor5	111.20	111.20	111.20	0.00	0.00	111.20	111.20
sensor6	98.60	98.60	98.60	0.00	0.00	98.60	98.60

Function: sparse_ Total Time(sec): 0.818452

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	92.00	93.00	1.00	1.00	93.00	93.00
sensor2	92.30	92.75	93.20	0.45	0.20	93.20	92.30
sensor3	102.20	103.10	104.00	0.90	0.81	104.00	104.00
sensor4	105.80	107.60	109.40	1.80	3.24	109.40	105.80
sensor5	111.20	111.20	111.20	0.00	0.00	111.20	111.20
sensor6	98.60	98.60	98.60	0.00	0.00	98.60	98.60

Function: makea_ Total Time(sec): 4.003958

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.00	91.00	0.00	0.00	91.00	91.00
sensor2	94.10	94.10	94.10	0.00	0.00	94.10	94.10
sensor3	102.20	102.20	102.20	0.00	0.00	102.20	102.20
sensor4	102.20	102.20	102.20	0.00	0.00	102.20	102.20
sensor5	109.40	109.43	111.20	0.23	0.05	109.40	109.40
sensor6	98.60	98.60	98.60	0.00	0.00	98.60	98.60

Table 5: Partial Tempest functional profile for CG benchmark with NP=4 and Class C.

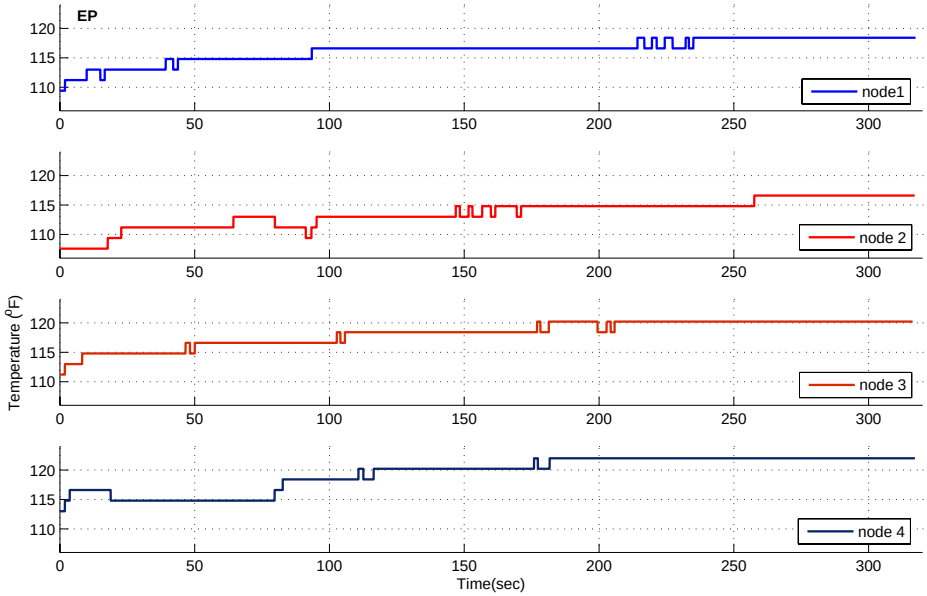

Figure 10: Thermal profile of EP benchmark, NP=4 , Class=C

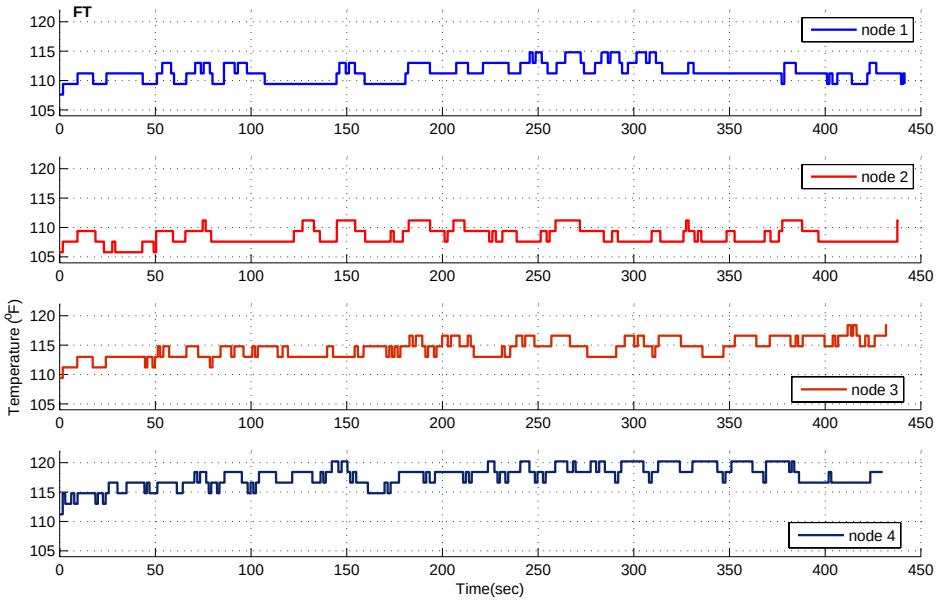

Figure 11: Thermal profile of FT benchmark, NP=4, Class=C

Function: vranlc_ Total Time(sec): 150.681946

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	93.00	93.01	95.00	0.14	0.02	93.00	95.00
sensor2	93.20	94.05	95.00	0.58	0.34	94.10	95.00
sensor3	103.10	103.59	104.90	0.51	0.26	104.00	104.90
sensor4	102.20	105.29	107.60	0.88	0.77	105.80	107.60
sensor5	107.60	113.56	116.60	2.36	5.57	114.80	116.60
sensor6	96.80	99.60	102.00	1.39	1.94	100.0	102.00

Function: randlc_ Total Time(sec): 0.046287

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	93.00	93.01	95.00	0.14	0.02	93.00	93.00
sensor2	93.20	94.05	95.00	0.58	0.34	94.10	95.00
sensor3	103.10	103.59	104.90	0.51	0.26	104.00	104.90
sensor4	102.20	105.31	107.60	0.87	0.76	105.80	107.60
sensor5	107.60	113.62	116.60	2.36	5.50	114.80	116.60
sensor6	96.80	99.62	102.00	1.39	1.93	100.00	102.00

Function: timer_clear__ Total Time(sec): 0.000001

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	93.00	93.00	93.00	0.00	0.00	93.00	93.00
sensor2	95.00	95.00	95.00	0.00	0.00	93.20	95.00
sensor3	104.00	104.00	104.00	0.00	0.00	103.10	104.00
sensor4	102.20	102.20	102.20	0.00	0.00	102.20	102.20
sensor5	107.60	107.60	107.60	0.00	0.00	107.60	107.60
sensor6	96.80	96.80	96.80	0.00	0.00	96.80	96.80

Table 6: Partial Tempest functional profile for EP benchmark with NP=4, class C.

Function: compute_indexmap__ Total Time(sec): 5.205775

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.45	93.00	0.84	0.69	91.00	91.00
sensor2	93.20	93.70	94.10	0.45	0.19	94.10	94.10
sensor3	102.20	103.09	104.00	0.56	0.35	103.10	103.10
sensor4	100.40	101.64	102.20	0.83	0.69	102.20	102.20
sensor5	111.20	114.55	116.60	2.34	5.52	114.80	111.20
sensor6	100.40	101.27	102.20	0.89	0.81	100.40	100.40

Function: vranlc_ Total Time(sec): 9.885297

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.82	93.00	0.98	0.96	91.00	93.00
sensor2	93.20	93.77	94.10	0.43	0.18	94.10	94.10
sensor3	102.20	102.88	104.00	0.41	0.17	103.10	103.10
sensor4	102.20	102.20	102.20	0.00	0.00	102.20	102.20
sensor5	113.00	114.32	116.60	1.08	1.17	114.80	114.80
sensor6	100.40	101.25	102.20	0.89	0.80	100.40	102.20

Function: transpose2_local__ Total Time(sec): 20.722044

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.53	93.00	0.88	0.78	91.00	93.00
sensor2	93.20	93.41	94.10	0.38	0.15	93.20	94.10
sensor3	102.20	103.25	104.00	0.39	0.16	103.10	104.00
sensor4	102.20	103.73	107.60	1.58	2.52	104.00	104.00
sensor5	114.80	118.49	120.20	1.81	3.23	118.40	116.60
sensor6	100.40	103.71	105.80	1.29	1.68	104.00	102.20

Table 7: Partial Tempest functional profile of FT benchmark with NP=4, class C.

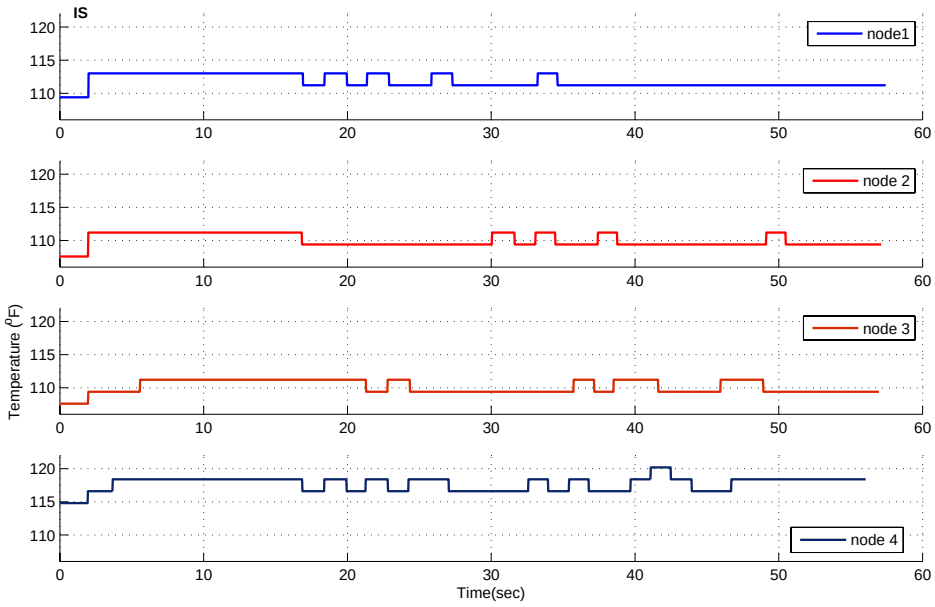

Figure 12: Thermal profile of IS benchmark, NP=4 and Class=C

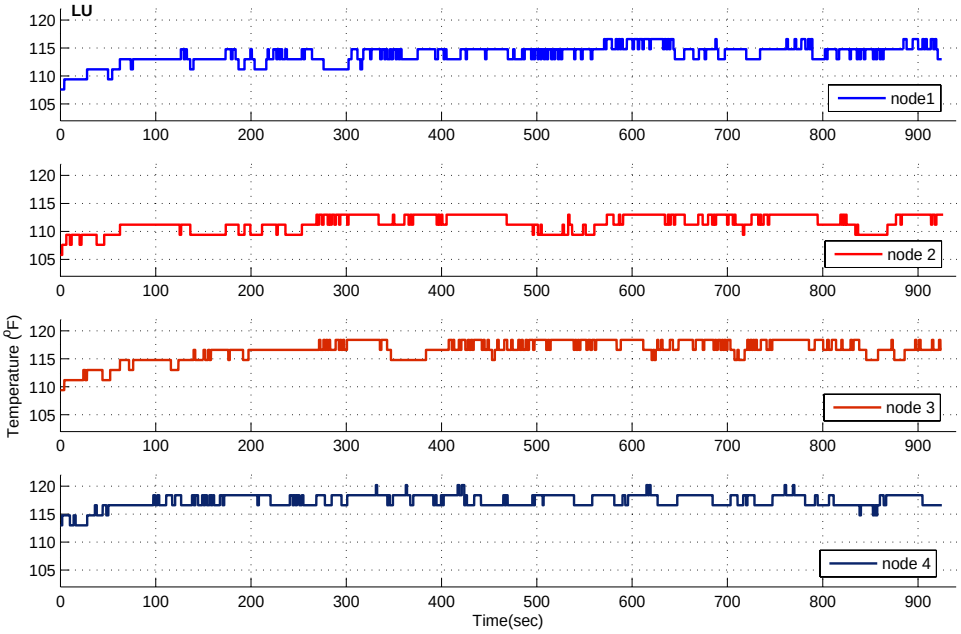

Figure 13: Thermal profile of LU benchmark, NP=4 and Class=C

Function: `find_my_seed` Total Time(sec): 0.000015

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	93.00	93.00	93.00	0.00	0.00	0.00	0.00
sensor2	93.20	93.20	93.20	0.00	0.00	0.00	0.00
sensor3	102.20	102.20	102.20	0.00	0.00	0.00	0.00
sensor4	104.00	104.00	104.00	0.00	0.00	0.00	0.00
sensor5	109.40	109.40	109.40	0.00	0.00	0.00	0.00
sensor6	100.40	100.40	100.40	0.00	0.00	0.00	0.00

Function: `randlc` Total Time(sec): 12.638560

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.99	93.00	1.00	1.00	93.00	93.00
sensor2	93.20	93.55	94.10	0.44	0.19	93.20	94.10
sensor3	102.20	102.58	103.10	0.45	0.19	102.20	103.10
sensor4	104.00	104.00	104.00	0.00	0.00	104.00	104.00
sensor5	109.40	112.53	113.00	1.21	1.47	113.00	113.00
sensor6	100.40	100.40	100.40	0.00	0.00	100.40	100.40

Function: `create_seq` Total Time(sec): 12.638553

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.99	93.00	1.00	1.00	93.00	93.00
sensor2	93.20	93.55	94.10	0.44	0.19	93.20	94.10
sensor3	102.20	102.58	103.10	0.45	0.19	102.20	103.10
sensor4	104.00	104.00	104.00	0.00	0.00	104.00	104.00
sensor5	109.40	112.53	113.00	1.21	1.47	113.00	113.00
sensor6	100.40	100.40	100.40	0.00	0.00	100.40	100.40

Table 8: Partial Tempest functional profile of IS benchmark with NP=4, class C.

Function: `exact_` Total Time(sec): 2.823563

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.80	93.00	0.98	0.96	91.00	91.00
sensor2	93.20	93.27	94.10	0.25	0.06	93.2 0	93.20
sensor3	102.20	103.17	104.00	0.84	0.71	103.10	102.20
sensor4	100.40	101.35	104.00	1.15	1.33	100.40	100.40
sensor5	105.80	107.29	113.00	2.01	4.05	107.60	105.80
sensor6	96.80	97.05	100.00	0.87	0.76	96.80	96.80

Function: `setiv_` Total Time(sec): 2.509814

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.89	93.00	0.99	0.99	91.00	91.00
sensor2	93.20	93.20	93.20	0.00	0.00	93.20	93.20
sensor3	102.20	103.16	104.00	0.88	0.77	104.00	102.20
sensor4	100.40	101.13	102.20	0.88	0.78	100.40	100.40
sensor5	105.80	106.82	109.40	1.15	1.32	105.80	105.80
sensor6	96.80	96.80	96.80	0.00	0.00	96.80	96.80

Function: `erhs_` Total Time(sec): 1.175513

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	92.93	93.00	0.36	0.13	93.00	91.00
sensor2	93.20	93.20	93.20	0.00	0.00	93.20	93.20
sensor3	102.20	102.26	104.00	0.32	0.10	102.20	102.20
sensor4	100.40	100.40	100.40	0.00	0.00	100.40	100.40
sensor5	105.80	107.51	107.60	0.39	0.15	107.60	105.80
sensor6	96.80	96.80	96.80	0.00	0.00	96.80	96.80

Table 9: Partial Tempest functional profile of LU benchmark with NP=4, class C.

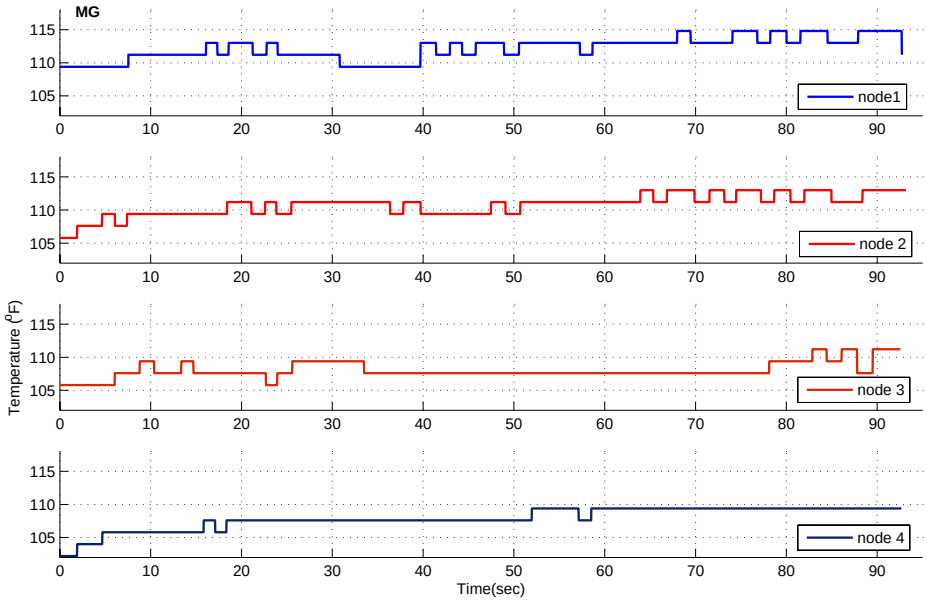

Figure 14: Thermal profile of MG benchmark, NP=4 and Class=C

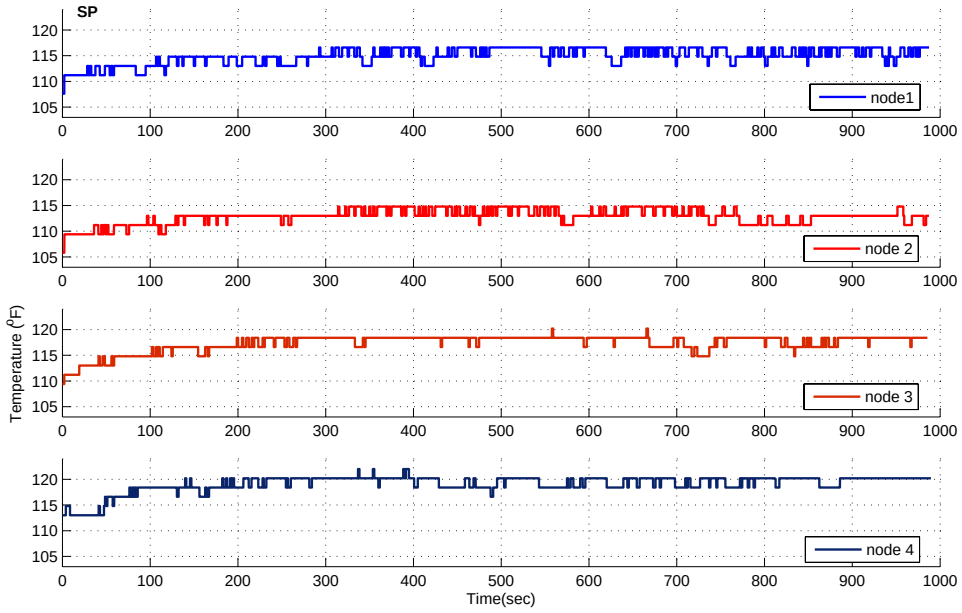

Figure 15: Thermal profile of SP benchmark, NP=4 Class=C

Function: psinv_ Total Time(sec): 22.743546

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.92	93.00	0.99	0.99	91.00	93.00
sensor2	93.20	93.57	94.10	0.44	0.19	93.20	94.10
sensor3	103.10	103.54	104.00	0.45	0.20	103.10	104.00
sensor4	105.80	108.27	109.40	1.17	1.39	107.60	109.40
sensor5	113.00	113.66	114.80	0.87	0.75	113.00	114.80
sensor6	102.20	102.20	102.20	0.01	0.00	102.20	102.20

Function: interp_ Total Time(sec): 5.603179

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	92.09	93.00	0.99	0.99	93.00	93.00
sensor2	93.20	93.64	94.10	0.45	0.20	93.20	94.10
sensor3	103.10	103.45	104.00	0.44	0.19	103.10	104.00
sensor4	105.80	108.14	109.40	1.25	1.57	107.60	109.40
sensor5	113.00	113.69	114.80	0.87	0.77	113.00	114.80
sensor6	102.20	102.20	102.20	0.00	0.00	102.20	102.20

Function: zran3_ Total Time(sec): 6.485692

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	91.33	93.00	0.74	0.55	91.00	91.00
sensor2	93.20	93.42	94.10	0.39	0.15	93.20	93.20
sensor3	103.10	103.65	104.00	0.44	0.19	104.00	104.00
sensor4	102.20	104.48	105.80	1.45	2.09	104.00	105.80
sensor5	113.00	113.00	113.00	0.00	0.00	113.00	113.00
sensor6	102.20	102.20	102.20	0.00	0.00	102.20	102.20

Table 10: Partial Tempest functional profile of MG benchmark with NP=4, class C.

Function: txinvr_ Total Time(sec): 21.652561

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	92.12	93.00	0.99	0.98	93.00	93.00
sensor2	93.20	93.79	95.00	0.64	0.41	94.10	95.00
sensor3	102.20	103.28	104.00	0.59	0.36	103.10	104.00
sensor4	102.20	103.19	105.80	1.15	1.33	102.20	104.00
sensor5	109.40	112.89	114.80	1.37	1.88	113.00	114.80
sensor6	96.80	99.66	100.00	0.79	0.63	100.00	98.60

Function: ninvr_ Total Time(sec): 36.279491

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	92.15	93.00	0.99	0.98	93.00	93.00
sensor2	93.20	93.76	95.00	0.64	0.41	94.10	95.00
sensor3	102.20	103.25	104.00	0.61	0.37	103.10	104.00
sensor4	100.40	103.13	105.80	1.17	1.38	102.20	105.80
sensor5	109.40	112.93	114.80	1.45	2.09	113.00	114.80
sensor6	96.80	99.62	100.00	0.85	0.72	100.00	98.60

Function: pinvr_ Total Time(sec): 29.338121

	Min	Avg	Max	Sdv	Var	Med	Mod
sensor1	91.00	92.15	93.00	0.99	0.98	93.00	93.00
sensor2	93.20	93.76	95.00	0.64	0.42	93.20	95.00
sensor3	102.20	103.26	104.00	0.61	0.37	103.10	104.00
sensor4	100.40	103.06	105.80	1.21	1.47	102.20	105.80
sensor5	109.40	112.99	114.80	1.49	2.24	113.00	114.80
sensor6	96.80	99.64	100.00	0.82	0.68	100.00	98.60

Table 11: Partial Tempest functional profile of SP benchmark with NP=4, class C.

The IS benchmark does show some signs of workload synchronization across nodes. This integer sort benchmark is susceptible to influence by the random nature of sort, but the thermal patterns are at least partially aligned as the sorts proceed across nodes. Again, despite the fact that the workload is distributed evenly across nodes, the temperature variance is significant across nodes from a low-max of about 112 degrees on node 2 to a high-max of almost 120 degrees on node 4.

The LU benchmark performs a parallel LU decomposition which is computationally intensive. This intensity appears to result in higher overall temperatures and result in a trend similar to the foo1 micro-benchmark of monotonically increasing temperatures over time. Once again, nodes 3 and 4 appear to be running hot. We could find no external reasons (ambient temperatures or position in the rack) to attribute to the observation that nodes 3 and 4 tend to run hot compared to the other node despite no dramatic workload differences.

The MG benchmark performs multi-grid calculations with regularity and some computational intensity. This code appears to contradict the trend of nodes 3 and 4 running hot. Here none of the nodes appear synchronized and each exhibits somewhat unique behavior. For example, node 4 is monotonically increasing but runs relatively cool staying below 110 degrees. Node 2 shows somewhat similar behavior with more peaks and valleys at higher frequencies. Node 3 appears to get warm then cool while node 1 grows steadily warmer. Though MG is load balanced it would appear the local workloads result in very different thermal behavior.

The SP benchmark solves scalar diagonal systems for finite element differences. This code is also computationally intensive and there does appear to be a significant warming trend as the benchmark progresses. Here, as observed earlier for computationally intensive codes, SP runs hot on nodes 3 and 4 at nearly 120 degrees. Nodes 1 and 2 hover around 115 degrees.

The EP benchmark seems to support our hypothesis that computationally intensive codes exhibit warming trend behavior. This embarrassingly parallel benchmark is computationally bound with very few communication events which leaves no opportunities for the CPU to cool as the benchmark progresses. Thus heat builds and as more power is dissipated without idle CPU time, the CPU core temperature grows. Nodes 3 and 4 run hot once again and all nodes show signs of warming over time. Nodes 2 and 3 manage to stay below 120 degrees.

The FT(Fourier Transform) benchmark was of great interest at first since it shows very regular behavior in its power profile [13]. Thus, we expected FT which spends 50% of its time in all-to-all communication to run fairly cool due to CPU idle time with noticeable trends. It is interesting to observe the thermal profile. What we see is no clear system wide trends in the thermals though the power trends are regular. Nodes 3 and 4 show steadily warming trends while nodes 1 and 2 have somewhat volatile behavior around an average (lower) temperature.

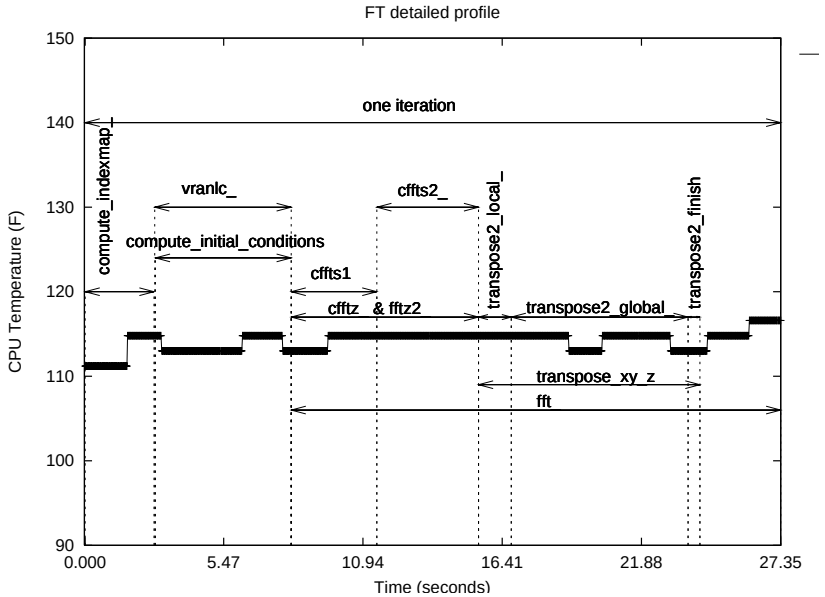

Figure 16: Fine-grain FT thermal profile, NP=4 and Class= C

The FT benchmark results were interesting that we used Tempest to study the function-level behavior in further detail to see if we could explain the trends. Figure 16 shows the fine-grain profile of a segment of an iteration for the FT benchmark on node 4 of the cluster. In this example, one can see the warming trend of this application from left to right as the iteration proceeds. Though the transpose operations are likely to be memory bound, they do not appear to slow the effects of the compounding heat on the processor. Recall, these temperatures are rising despite the fact that the fan speed is set to a static high speed.

We also studied the thermal variance across two different processors running SPEC cpu2000 sequential benchmark. Figure 17 illustrates the results on AMD 3200+ and 4000+ processors. The overall thermal trends are identical for both the CPU's.

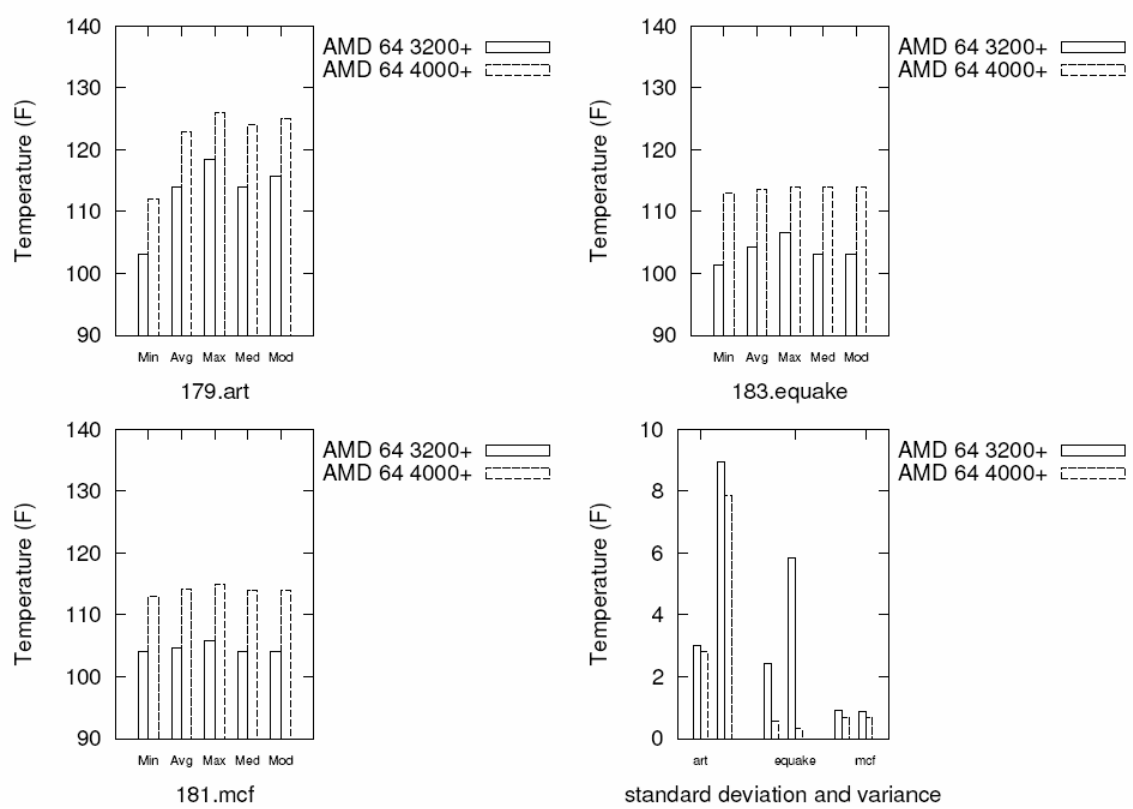

Figure 17: Illustrates thermal variance for some of the SPEC cpu 2000 benchmarks. The overall thermal trends seem to be identical for both the processors.

In this section we validated Tempest profiler using a wide range of scientific benchmarks. Tempest provides previously unavailable insight into the thermal behavior of applications. Tables 4 through 11 clearly demonstrate that Tempest is capable of providing fine-grain timing and thermal information transparently. This information is useful in addressing many research challenges discussed in Chapter 1 including identifying hotspots in codes, when and where to begin thermal optimizations etc. Furthermore, our results show that these characteristics may vary with the type of processor, architecture, placement of onboard sensors, fans etc. In subsequent sections we present some of the results of Tempest PID controller and their implications. We also discuss the effects of simple loop optimizations on the amount of heat dissipated.

Using Tempest PID controller for Parallel Benchmarks

As a part of our experimental evaluation, we used some common application suites from the parallel community to benchmark cluster systems. For brevity, we focus on codes from the NAS parallel benchmark suite. Figures 18 through 25 show the temperature regulation for eight of the NAS PB codes. Each set of graphs for a single benchmark code includes measurements of CPU temperature in Fahrenheit and time of execution for both the default case (top graphs per code for no DVFS running at highest frequency) and our temperature-aware controller case (bottom graphs per code using DVFS). We ran NAS parallel benchmark suite run with critical_thresholds of 110F and 105F and minimum_thresholds of 105F and 100F. Notice that the controller runs below the default with a performance overhead of under 10%.

The EP benchmark when run unregulated shows an increasing trend in temperature. The controller however heuristically tries to curtail this increasing temperature and regulate it close to under 110F. The PID controller modulates the processor frequency at runtime based on a predicted temperature. This is seen in Figure 20, at first the processor is scaled down to a 2.0GHz, however since EP is thermally active, the CPU is further scaled to 1.8GHz in order to restrain any further increase in temperature.

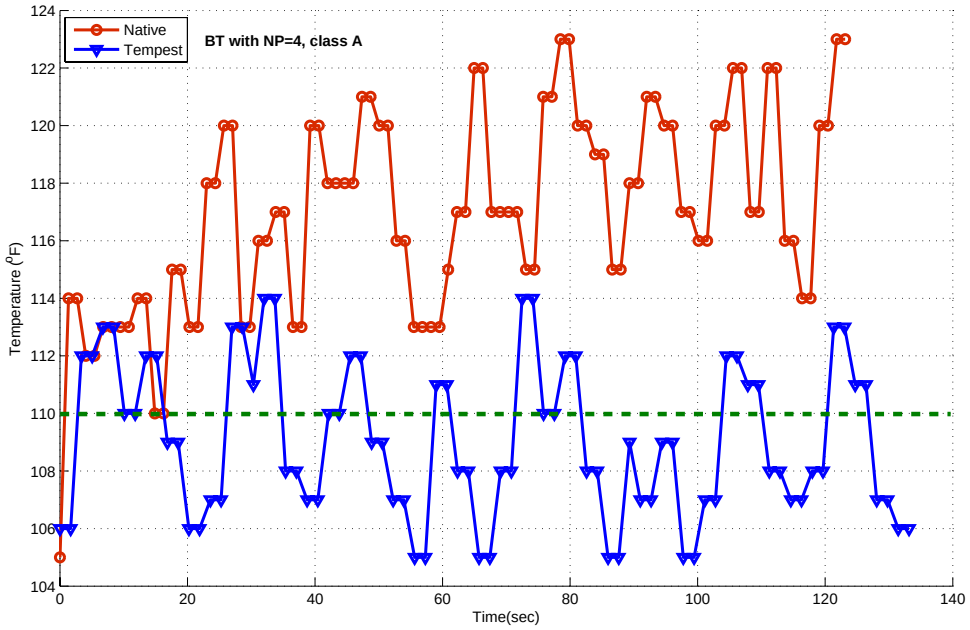

Figure 18: (above) Tempest PID controller reduces thermals for BT benchmark, NP=4 Class=A

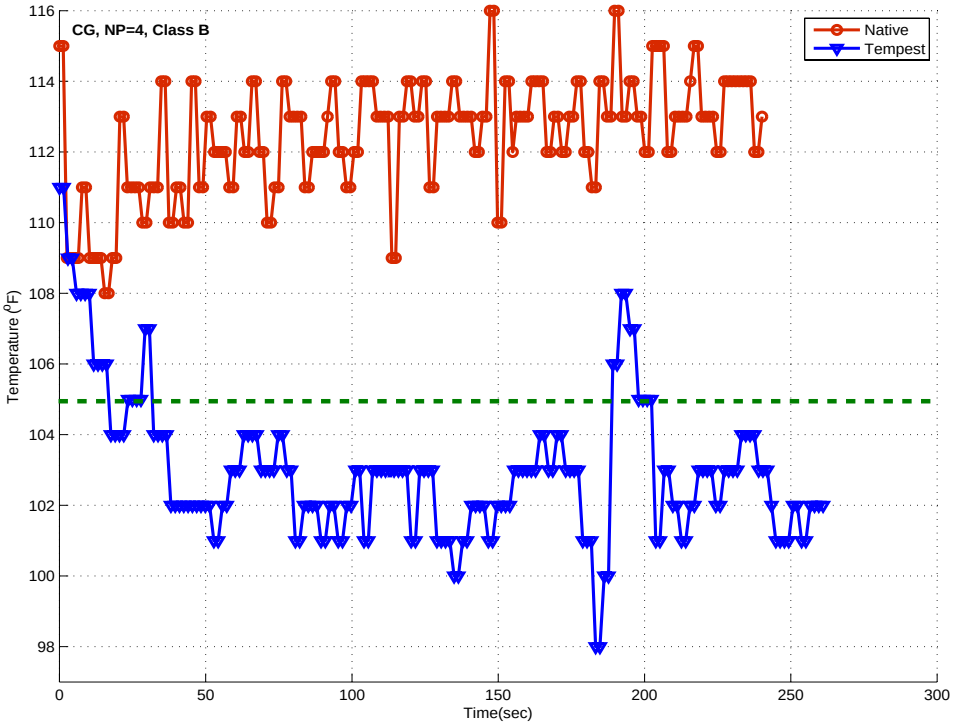

Figure 19: Tempest PID controller reduces thermals for CG benchmark, NP=4 Class=B.

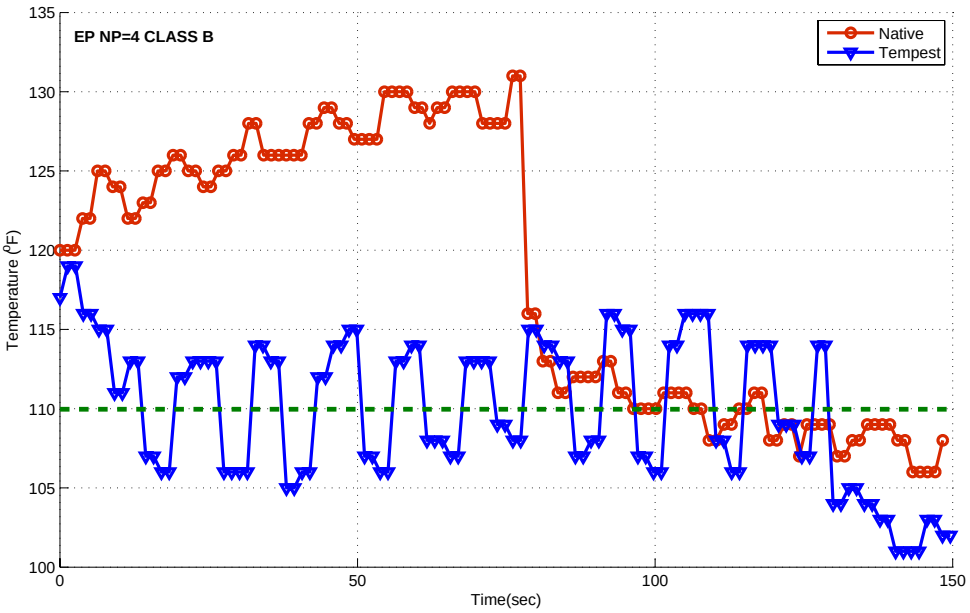

Figure 20: (above) Tempest PID controller reduces thermals for EP benchmark, NP=4 Class=B.

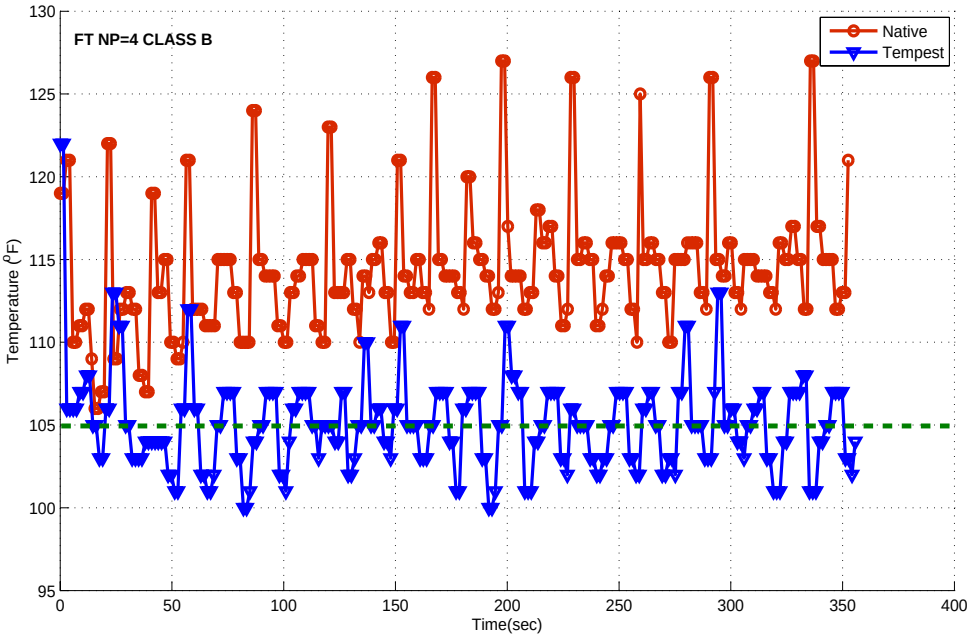

Figure 21: Tempest PID controller reduces thermals for FT benchmark, NP=4 Class=B.

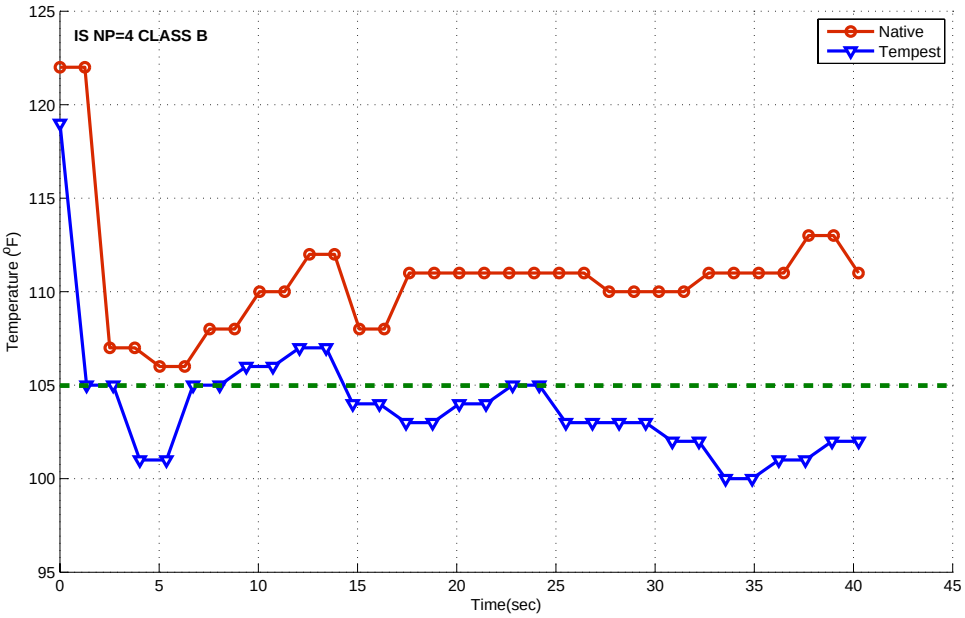

Figure 22: (above) Tempest PID controller reduces thermals for IS benchmark, NP=4 Class=B.

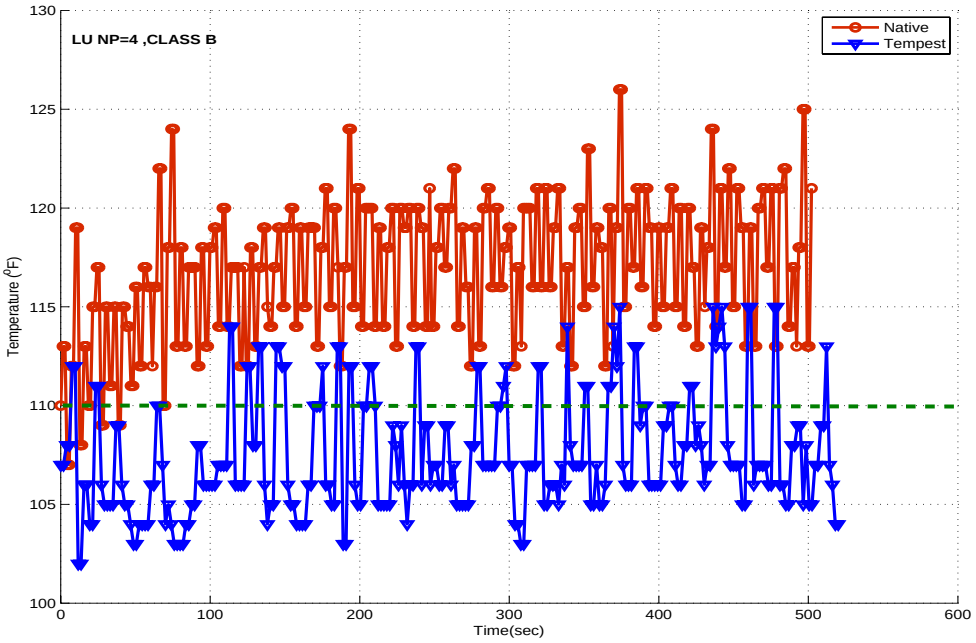

Figure 23: Tempest PID controller reduces thermals for LU benchmark, NP=4 Class=B.

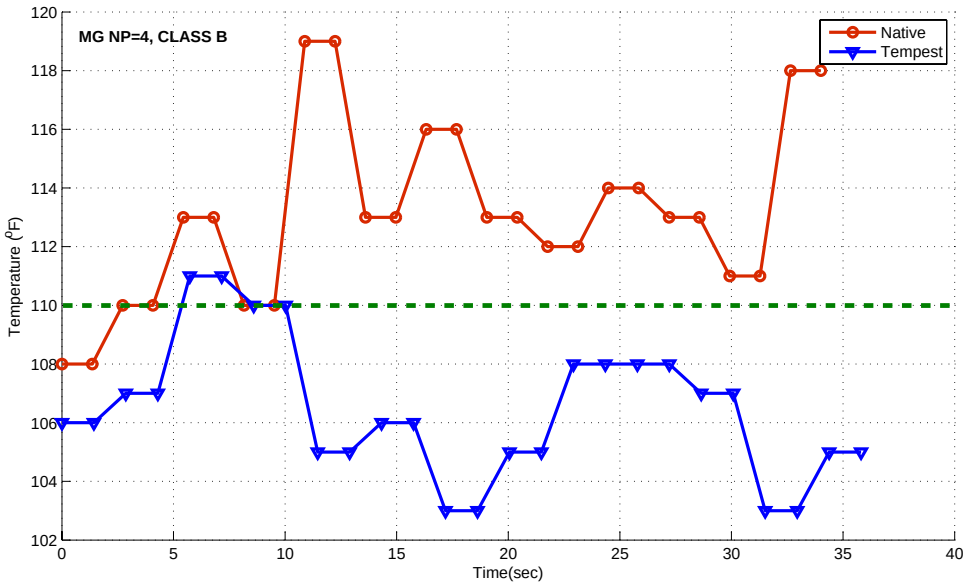

Figure 24: (above) Tempest PID controller reduces thermals for MG benchmark, NP=4 Class=B.

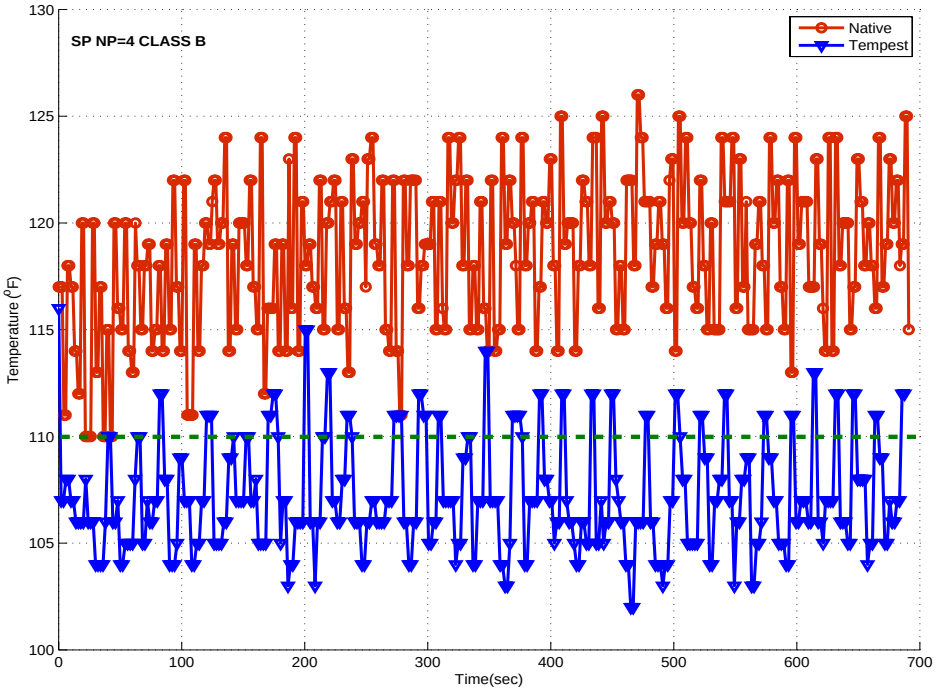

Figure 25: Tempest PID controller reduces thermals for SP benchmark, NP=4 Class=B.

The parallel CG benchmark uses under 30% of the CPU and little over 512KB of memory for class B workload. The critical threshold temperature is set to 105F. CG also shows an increasing steady slope. The PID controller executes the application at an average temperature of 103F most of the time.

The IS benchmark does not show signs of significant use of CPU. This integer sort benchmark uses under 20% of CPU and little over 32KB of main memory. The benchmark shows a moderate increase in temperature and the controller also regulates the temperature to stay around105F. As a result, the time of execution of the benchmark with and without the use of the controller is identical.

The LU benchmark performs a parallel LU decomposition which is computationally intensive. LU uses less than 32KB of memory. This intensity appears to result in higher overall temperatures over time. The controller regulates the temperature as shown in Figure 23. The temperature is curtailed to 110F on an average by controlling the CPU frequency.

The SP benchmark solves scalar diagonal systems for finite element differences. This code is also computationally intensive and therefore results in a significant warming trend as the benchmark progresses. Here, as observed earlier for computationally intensive codes, the controller maintains the temperature below the critical value, by scaling the CPU frequency as and when required. The controller regulates this increasing temperature to under 110F on an average.

The MG benchmark performs multigrid calculations with some computational intensity. The code runs relatively cooler with relatively less memory and CPU usage. The controller averts the CPU from reaching peak performance by controlling the processor frequency. The temperature is maintained under the critical threshold for most of the execution time.

The BT benchmark performs several tasks followed by a set of synchronization events that occur at regular intervals. However, like most of the benchmarks in the suite, BT

shows increasing trends in temperature. As a result, as shown in Figure 18 this increase is minimized by the controller by scaling the frequency. The controller tries running BT at an average temperature of under 110F showing a distinct difference in thermals between the native and the controller with minimal impact on performance.

The FT benchmark shows an increasing and decreasing trend in temperature because of its inherent computation and communication phases. However the controller instrumented code significantly lowers the temperature by at least 18F. This FT parallel benchmark is computationally bound with communication events which leaves opportunities for the CPU to cool and heat as the benchmark progresses. As a result, heat builds and as more power is dissipated without idle CPU time, the CPU core temperature grows and in order to counter this, the controller scales down CPU frequency.

Next we lowered our thermal thresholds to gauge the impact on performance. For some set of experiments, we lowered the critical_temperature to below 100F. We observed no significant differences in temperature (3-10 degrees Fahrenheit lower on average) with performance loss exceeding over 10% for some benchmarks, indicating the lower bounds on critical temperature threshold. We have so far discussed the thermal effects of the PID controller. In the following section we will discuss its implications on the overall performance of the benchmarks.

Performance implications of PID Controller

Figure 26 illustrates the performance implications of using the PID controller. It can be observed from previous discussion that the use of PID controller reduces the temperature by 10C with minimal overhead of under 10% and maintain the temperature within a desired operating range. We comprehensively observed and measured the runtime effects of processor frequency on thermals. From these figures we conclude that setting thermal thresholds can significantly reduce thermals without affecting performance as drastically as previously thought. Our hope is that with further experimentation of various controller policies we will be able to constrain the performance loss further while reducing the thermals significantly.

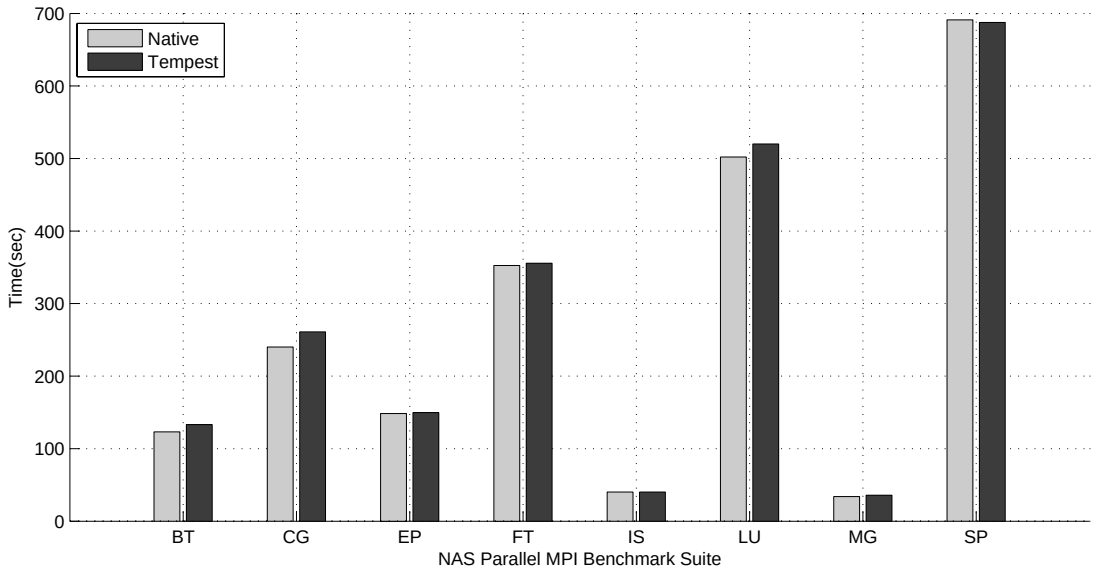

Figure 26: Performance implications of Tempest PID controller.

We also studied the effects of loop and cache optimizations using synthetic matrix multiplication benchmark. Figure 27 illustrates that optimizations such as cache blocking and loop unrolling dissipates lesser amounts of heat compared to other thermal optimizations.

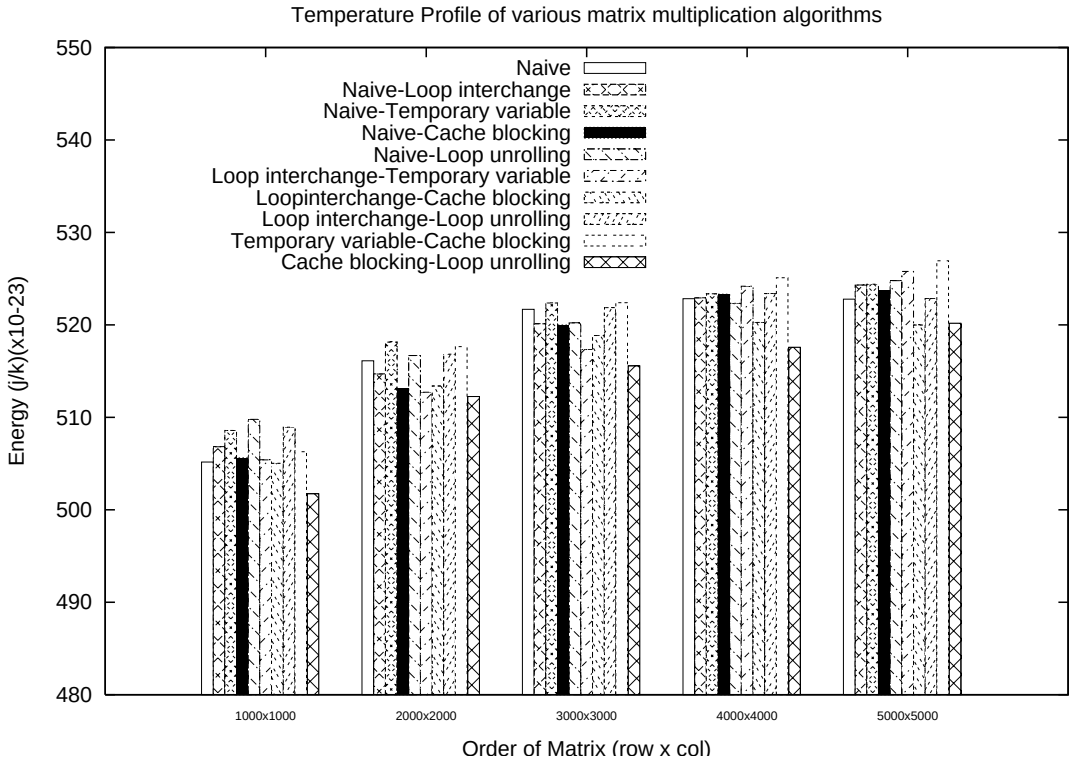

Figure 27: Illustrates the average heat dissipation based on $E = kT$, where T is the absolute average temperature and *k* is Boltzmann constant. Clearly loop optimizations and cache blocking have an impact on the overall heat generated.

Typically, such optimizations are done from a performance perspective. However sometimes such optimizations offer added advantages such as decreased thermals. These results help in designing energy and thermal efficient compilers.

In this section we have presented some of our experimental results with Tempest framework. This work shows the viability of thermal-aware high-performance computing as a promising new research path. Our hope is that with further experimentation and optimizations to Tempest we will be able to reduce the instrumentation overheads and provide improved cooling techniques to other onboard components.

Chapter 5

Concluding Remarks

This thesis has proposed the Tempest Framework for parallel and sequential scientific applications. We have designed and implemented a *middleweight* framework for fine-grained profiling of the thermal properties of parallel scientific applications. We demonstrated that though our framework is capable of granularity beyond the popular gprof performance tool, we have achieved comparable (if not better) overheads in our profiling. We showed how Tempest can profile applications, reduce the temperature by 10C with minimal overhead of under 10%, and maintain the temperature within a desired operating range.

We used the framework to perform the first fine-grain thermal profiling of parallel scientific applications running on real systems. Our results indicate that consideration of the thermal properties of systems and applications is a complicated endeavor. The workload characteristics including amount and type of computation can affect the thermals significantly while there is also variance observable for the same workload across different nodes in a system. We used Tempest to identify hot nodes and hot spots in code for several benchmarks.

We used Dynamic Voltage and Frequency Scaling (DVFS) to reduce the thermals of high-performance clusters by reducing processor voltage. We designed and developed a heuristic approach to regulate the temperature of the CPU. We comprehensively studied and measured the runtime effects of processor frequency on thermals. We found HPC

workload characteristics greatly impact the effects of DVFS on temperature. We identified other approaches that can maintain temperature thresholds and/or reduce temperature with minimal impact on performance. We provide means to improve reliability by regulating thermals.

We then discussed how Tempest can be used to evaluate the effects of thermal management techniques on the temperature in a cluster system. We tested for the effects of fan speed control and DVFS in isolation and in collaboration. Our results indicate that fan speed control, DVFS and loop optimizations and cache blocking can be effective means of reducing heat in a system. We believe these techniques used in unison twitters reduction in thermals.

In this thesis, we provided an infrastructure that addresses the research challenges discussed in Chapter 1. Tempest helps provide answers to questions such as where to optimize code to reduce thermals, what the possible optimizations and their implications are on performance, etc. Tempest demonstrates that by setting thermal thresholds, one can significantly reduce thermals without affecting performance as drastically as previously thought. Our hope is that with further experimentation of various controller policies we will be able to constrain the performance loss further while reducing the thermals significantly.

Salient contributions

The key contribution of the Tempest Framework lies in its ability to facilitate low-overhead runtime thermal profiling. This contribution allows both transparent and non-transparent source-code-level thermal profiling of applications written in C, C++ and/or FORTRAN. Tempest provides previously unavailable insight into the thermal characteristics of applications running on real systems.

The second contribution of the Tempest Framework lies in its ability to transparently regulate temperature of CPU facilitate with minimal overhead. This contribution provides means to improve reliability of CPU and other onboard components.

Together, these contributions facilitate high-performance-thermal-aware-distributed computing by optimizing for thermals, performance, and power. Ultimately, these contributions can improve the overall reliability and reduce power consumptions in large scale clusters.

The Tempest framework helps provide a deeper insight into the thermal behavior of applications as demonstrated by experiments with commonly used parallel and sequential benchmarks. These experiments in two different contexts (desktop and high performance arenas) accentuate the broad impact of Tempest framework. The results from the experiments of sequential benchmarks on desktop environments show that it is possible to attribute the thermals of a system at the granularity of a few lines of code in a process. The results from the experiments of parallel benchmarks in clusters indicate that it is possible to profile and regulate parallel applications for thermals, and contribute to improved reliability and reduced mean failure rate.

These results together demonstrate that the Tempest framework can provide an insight into the thermal characteristics of scientific applications by exploiting hardware sensors. Ultimately, sensitive and more reliable sensors enable more accurate information while profiling. Advances in sensor technology can substantiate the usefulness of the Tempest framework.

Other Aspects

The Tempest framework offers its services as a shared library that supports simple APIs. Users can explicitly profile and control their applications using these APIs. The framework also provides meta-language for specifying configurations for profiling.

Another noteworthy aspect of Tempest Framework is its current implementation. Currently, the Tempest framework supports five architectures: x86, x86_64, CMP, SMP and Power PC. However, since most of the Tempest framework is fairly architecture-neutral, the implementation can be easily extended to other architectures like Sparc, Digital Alpha, and Cell. Porting to different operating system like Windows and OS-X is also easily achievable. Since, most operating systems support hardware sensors, allow means to bind a processes to cores and provide fine grained timing information, theoretically, the Tempest framework is portable across a wide range of architectures and operating systems.

Tempest scales well with number of nodes in a cluster. In fact, each Tempest-aware process is bound to a core or a processor in a node. The profiling and controlling is done on a per-node basis. Hence, the number of nodes in a cluster does not impinge the performance of Tempest libraries. Generally, we found Tempest to be portable and provide accurate, repeatable measurements.

Chapter 6

Ongoing and Future work

Ongoing work

In our studies so far, we developed infrastructure to analyze the applications for thermals. We are presently studying the causes of thermal variations in applications. In our preliminary experiments we identified events such as L1 and L2 cache misses, branch mispredictions etc. contribute to increased CPU heat dissipation. Previous research through simulation has found out that floating point and register file operations often cause an increase in CPU temperature. Essentially, this work intends to use hardware counters and profile the applications for events such as cache misses etc. and correlate such events at source-level to temperature. This approach provides a deeper insight into the causes of thermal behavior of applications and also directs to improving the performance of the application.

Future work

Though we have obtained promising results, more work is needed. We have only begun to use Tempest to study interesting thermal phenomena in clusters. Though we have some understanding of the trends in thermals for various workloads, we need to isolate performance characteristics at finer granularity to see if we can identify specific traits in codes that lead to higher thermals. These kinds of observations could lead to techniques that encourage thermal-aware code (or library) development. We would also like to study

the impact of other management techniques such as cluster-wide workload migration from hot servers to cooler servers. Though this has been done for commercial workloads, the insight provided by Tempest could identify tradeoffs between various techniques that have not been identified. Also, we would like to study the use of Tempest data at runtime to make thermal management decisions.

We would also like to quantify the effects of DVFS on reliability. We are presently investigating a multidimensional unified suite that addresses power, energy, performance and thermal issues in large scale clusters. In our future work, we would also like to study other thermal management techniques such as process migration and load balancing. We would also like to port Tempest to other architectures like Sparc and Cell. We plan to accomplish all our goals based on real systems without resorting to simulation.

We would also like to develop an infrastructure to manage the results obtained from several nodes in a cluster. A graphical interface that supports options pertaining to presenting results in a wide variety of formats would help in understanding the overall thermal behavior of the cluster.

References and Bibliography

1. Koomey, J. Estimating total power consumption by servers in the US and the world. in EPA Data Centers Technical Workshop. 2007. Santa Clara, CA.

2. lm_sensors. 2007 [cited 2007 05/28/07]; Available from: http://www.lm-sensors.org/.

3. multimeters. 2007 [cited 2007 05/28/07]; Available from: http://www.multimeterwarehouse.com/.

4. Brooks, D. and M. Martonosi, Dynamic Thermal Management for High-Performance Microprocessors, in Proceedings of the 7th International Symposium on High-Performance Computer Architecture. 2001, IEEE Computer Society.

5. Brooks, D., et al., Power-Performance Modeling and Tradeoff Analysis for a High End Microprocessor, in Proceedings of Workshop on Power-Aware Computer Systems(PACS2000, held in conjunction with ASPLOS-IX). 2000: Cambridge, MA.

6. Huang, M., et al., A framework for dynamic energy efficiency and temperature management, in Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture. 2000, ACM Press: Monterey, California, United States.

7. Li, Y., et al., Performance, Energy, and Thermal Considerations for SMT and CMP Architectures, in Proceedings of the 11th International Symposium on High-Performance Computer Architecture. 2005, IEEE Computer Society.

8. Liu, P., et al., Fast thermal simulation for architecture level dynamic thermal management, in Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design. 2005, IEEE Computer Society: San Jose, CA.

9. Skadron, K., M.R. Stan, and W. Huang. Temperature-Aware Microarchitecture. in proceedings of The 30th Annual International Symposium on Computer Architecture. 2003. San Diego, California.

10. Skadron, K., et al., Temperature-aware microarchitecture: Modeling and implementation. ACM Trans. Archit. Code Optim., 2004. **1**(1): p. 94-125.

11. Fluent. [cited 2006 26 Dec]; Available from: www.fluent.com.

12. Heath, T., et al., Mercury and freon: temperature emulation and management for server systems. SIGARCH Computer Architecture News, 2006. **34**(5): p. 106-116.

13. Cameron, K.W., R. Ge, and X. Feng, High-performance, power-aware, distributed computing for scientific applications. IEEE Computer, 2005. **38**(11): p. 40-47.

14. Freeh, V.W., et al. Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster. in 10th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP). 2005.

15. Brooks, D., et al. Power-Performance Modeling and Tradeoff Analysis for a High End Microprocessor. in Workshop on Power-Aware Computer Systems (PACS2000, held in conjuction with ASPLOS-IX). 2000. Cambridge, MA.

16. Skadron, K., et al., Temperature-Aware Microarchitecture: Modeling and Implementation. ACM Transactions on Architecture and Code Optimization, 2004. **1**(1): p. 94-125.

17. Skadron, K., et al. Temperature-Aware Microarchitecture. in The 30th Annual International Symposium on Computer Architecture. 2003. San Diego, California.

18. GNU Compiler. 2007 [cited 2007 05/29/07]; Available from: http://gcc.gnu.org/.

19. gprof. 2007 [cited 2007 05/29/07]; Available from: http://www.gnu.org/software/binutils/.

20. Skadron, K., et al., Temperature-aware computer systems: Opportunities and challenges. Micro, IEEE, 2003. **23**(6): p. 52-61.

21. Hung, W.L., et al. Thermal-aware task allocation and scheduling for embedded systems. 2005.

22. Chaparro, P., J. Gonzalez, and A. Gonzalez. Thermal-Effective Clustered Microarchitectures. in First Workshop on Temperature-Aware Computer Systems. 2004. Munich Germany.

23. Heath, T., et al., Mercury and freon: temperature emulation and management for server systems. SIGARCH Comput. Archit. News, 2006. **34**(5): p. 106-116.

24. Gurumurthi, S., et al. DRPM: Dynamic Speed Control for Power Management in Server Class Disks. in proceedings of ISCA. 2003.

25. Gurumurthi, S., A. Sivasubramaniam, and V. Natarajan. Disk Drive Roadmap from the Thermal Perspective: A case study for Dynamic Thermal Management. in proceedings of ISCA. 2005.

26. Sharma, R., et al., Balance of Power: Dynamic Thermal Management for Internet Data Centers. IEEE Internet Computing, 2005. **9**(1).

27. Bellosa, F. The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems. in Proceedings of 9th ACM SIGOPS European Workshop. 2000. Kolding, Denmark.

28. Merkel, A., F. Bellosa, and A. Weissel. Event-Driven Thermal Management in SMP Systems. in Second Workshop on Temperature-Aware Computer Systems. 2005. Madison, Wisconsin.

29. Moore, J., et al. Making Scheduling "Cool": Temperature-Aware Workload Placement in Data Centers. in USENIX 2005 Annual Technical Conference. 2005.

30. Chase, J.S., et al., Managing energy and server resources in hosting centers, in Proceedings of the eighteenth ACM symposium on Operating systems principles. 2001, ACM Press: Banff, Alberta, Canada.

31. Moore, J., J. Chase, and P. Ranganathan. ConSil: Low-Cost Thermal Mapping of Data Centers. in proceedings of SysML. 2006.

32. Thermal-Tempest. 2007 [cited 2007 06/06/07]; Available from: http://sourceforge.net/projects/thermal-tempest/.

33. GNU-glibc. glibc. 2006 [cited 2006 27th December 2006]; Available from: http://www.gnu.org/software/libc/.

34. cpufreq. 2006 [cited 2006 10/01/2006]; Available from: http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html.