Tempest: A portable tool to identify hot spots in parallel code

Kirk W. Cameron, Hari K. Pyla, and Srinidhi Varadarajan
Virginia Polytechnic Institute and State University
{cameron, harip, srinidhi}@cs.vt.edu

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. Lastly, 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 tool called Tempest (for temperature estimator) that leverages emergent thermal sensors to enable user profiling, evaluating, and reducing the thermal characteristics of systems and applications. In this paper, we illustrate the use of Tempest to analyze the thermal effects of various parallel benchmarks in clusters.

1. Introduction

The power consumption of clusters of servers has reached critical levels. The US Environmental Protection Agency recently announced its intention to reward energy efficient server designs with EnergyStar ratings, a program popularized for appliances and monitors in the last 25 years [9]. Server power 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.

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 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 thermal simulation tools [2, 7, 10, 13] are primarily designed to provide a means of estimating the thermals of proposed hardware configurations. Profiling data from simulation is extremely detailed. 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 scientists, mechanical and electrical engineers, and experts in computational fluid dynamics and computer systems and architecture [6]. Small system changes may require redesign and revalidation of the thermal model. Additionally, the time necessary to obtain simulated

[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.

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?
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?

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, and 4. Heavy-weight tools can provide the detail necessary, but many thermal simulators do not run native system software which may lead to inaccuracy. Portability to several systems quickly is also questionable due to thermal model development requirements while 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 1-4.

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

Tempest leverages existing GNU compiler support to transparently create thermal and performance profiles for any source code using simple command line executions. 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 also provides a non-transparent profiling library independent of the compiler.

Tempest does not significantly alter the performance of an application and incurs less than 7% overhead for temperature and performance profiling for the applications we studied. Tempest is portable and easy to use, allowing experts and non-experts alike to profile temperature and validate thermal reduction techniques.

There are several contributions in our work:

- *We propose, implement, and validate Tempest, a middle-weight tool 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 demonstrate how Tempest can be used to profile and analyze the effects of thermal optimizations on a parallel application.*

This paper is organized as follows. First we describe 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 we describe the design and validation of Tempest. Then we discuss the profiling results for several parallel benchmarks followed by an analysis of the thermals at function-level granularity. We conclude with a discussion of the limitations and future work for Tempest.

2. Related Work

To the best of our knowledge there have been no studies of the thermal properties of parallel scientific applications. 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. In 2003, Skadron et al proposed a simulation-based approach called HotSpot [14] and made the case for thermal considerations early in microprocessor layout design. Unfortunately, ascertaining the effects of heat dissipation can be cumbersome. HotSpot and tools proposed by others [4, 7, 8] 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 [6] that provides system-level thermal profiling data using a model-based emulation approach. They use discrete heat flow models 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, migrate tasks, etc.

Bellosa et al [1, 11] 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 the 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 [5, 12]. 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.

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 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.

3. Tempest Design and Framework

To profile the thermal characteristics of parallel scientific applications, we created a framework called Tempest (***temp***erature ***est***imator). Figure 1 illustrates how Tempest measures real time thermal data from individual nodes in a cluster using hardware sensors and correlates this data with program execution. While Tempest requires source code recompilation presently, runtime profiling and analysis is automatic and transparent. Using Tempest we are able to profile any function's thermal characteristics in a clustered system. In this section, we provide a high-level view of the inner workings of Tempest. Moreover, Tempest also provides API for a thermal-aware user to explicitly reference from within a program.

3.1 Design evolution

We had several design goals for Tempest. First, since we had experience using simulation-based thermal measurement tools like HotSpot, 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.

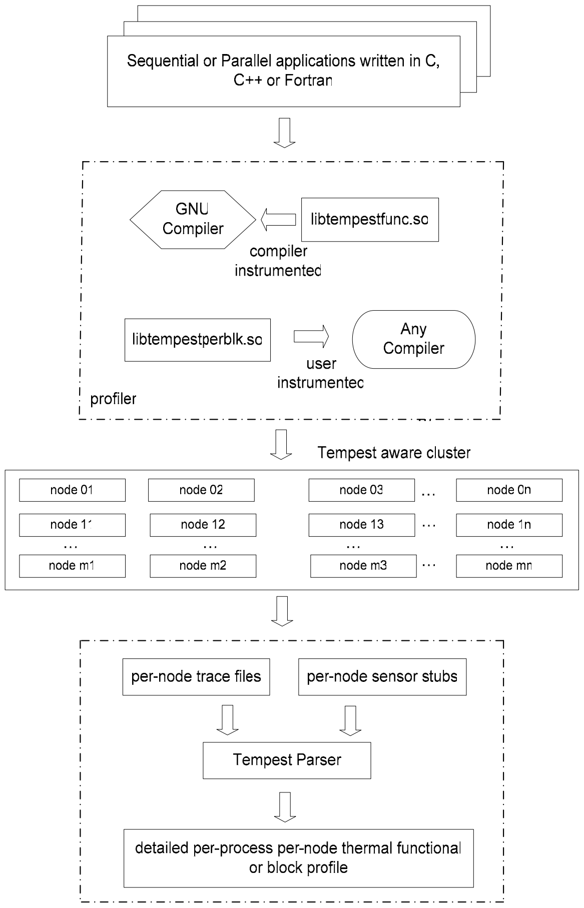


Figure 1: Tempest Framework Design. Tempest leverages the instrumentation provided in a GNU compiler to thermally profile an application on any number of nodes in a cluster. Users must simply compile with instrumentation enabled, link to one or more Tempest libraries, run their code, and invoke the Tempest parser for post processing. By default, Tempest writes data to the standard output, but data can be dumped to a file in a variety of formats.

We iterated through several designs before converging on our current implementation of Tempest. 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 functions at a very high level of granularity. For example, gprof creates buckets for functions and adds to buckets as it spends time in various functions; gprof does not pinpoint which function was executing at time X in a program. Tempest requires a function 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 glibc. We plan to reconsider both of these approaches in succeeding versions of the tool. Consequently, we adopted the approach of relying on the compiler to provide instrumentation hooks for tracing the execution of the program and a user-level library that is easily portable and can be linked to the application at compile time to generate run-time thermal measurements.

3.2 Implementation details

Our approach involves: 1) measuring the entry and exit times of a function; and 2) measuring the temperature during the course of a function's execution. We implemented a shared library that leverages support from the gcc compiler. The gcc compiler exports function-handlers to applications compiled with -finstrument-functions option. Using the 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 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 for sampling a hardware counter using the rdtsc instruction[2].

In order to measure temperature while the program is running, we created a light weight temperature measuring process (tempd). The tempd process samples temperature four times per second using sensors on the motherboard and processor. The tempd process is a part of our shared library and is launched before the main function of the profiled application is invoked. The profiling information for every node in the cluster along with the timestamps is aggregated into a trace file. Upon starting an application and just prior to exiting, the destructor in the shared library is called which sends a signal to tempd for termination and performs cleanup operations. In this implementation we assume that the underlying temperature sensors are accurate. We validated the hardware thermal sensors for accuracy by running a set of CPU intensive micro-benchmarks and comparing sensor measurements to those measured by an external sensor attached to the CPU.

The Tempest parser acquires function timestamps and provides a mapping between timestamps and temperature for the workload on the cluster. 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. We note that while function-level granularity is the focus of this paper, Tempest also supports measurement at basic block granularity using libtempestperblk.so. Basic block measurement is non-transparent and requires explicit API calls.

3.3 Limitations of Tempest

Tempest was designed to incur minimal profiling overhead. The rdtsc instruction minimizes sampling overhead, but introduces complications such as clock skewing across processors or cores. Tempest compensates for such issues by binding applications to a processor and core for the duration of execution and has been validated for the applications described on multi-processor and multi-core systems. Applications that consistently migrate processes across cores and processors would incur additional overhead and probably require modifications to Tempest for accurate results. Tempest also will incur additional overhead when profiling applications which invoke functions with very short life spans repeatedly. We are attempting to improve Tempest for such codes presently.

3.4 Verification

We tested Tempest on a number of systems and compilers first using a series of micro-benchmarks. Tempest measures data from all available thermal sensors. On the systems we measured, we observed as few as 3 sensors on x86 platforms from AMD and

[2] Technically speaking, rdtsc is platform dependent. However, we identified the equivalent instruction set on the PowerPC architecture.

Table 1: One of five micro-benchmarks to test Tempest correctness for various interleaving and recursion conditions. All benchmarks include: A (main alone), B (one function), C (multiple functions), D (multiple functions with interleaving), and E (multiple functions with recursion and interleaving)

```
D: main()
     {
     foo1(){
     foo2();
     }
     foo2();
     }
```

up to 7 sensors on PowerPC G5 systems. Tempest will run on any Linux-based system that has support for the LM sensors package. As mentioned, Tempest currently supports gcc for C code, g++ for C++ code, and g77 for FORTRAN code. Due to space limitations we present a subset of results we gathered using Tempest[3].

We compared Tempest measurements to gprof. The gprof tool provides an estimate of the time spent in each function of the program. We compared the original code, to the original code using gprof, and the original code using Tempest. Both tools provided similar results for total execution time in the various code functions within the variance mentioned.Gprof introduced less than 10% overhead to the original code for all codes measured including the SPEC CPU 2000 benchmarks and the NAS Parallel Benchmark suite. Tempest introduced less than 7% overhead for the same codes. Repeated measurements were subject to variance of about 5%. The results presented are an average sample from at least 5 runs.

4. Thermal Profiling Using Tempest

Tempest provides previously unavailable insight into the thermal characteristics of applications running on real systems. In this section we profile select results from various system and code implementations. Generally, we found Tempest is portable and provides accurate, repeatable measurements. Due to space limits, we do not report all sensors for all systems. We found the ambient sensors located throughout the system chassis valuable did not correlate significantly to source code phases and were more a reflection of external temperatures and airflow. Hence, we report only results from the core CPU sensors and find that the thermals of an application have some basic trends that reflect the phases of the application. Another interesting observation is that thermals vary between systems (under the same load) at times significantly.

4.1 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, a Linux package that allows system-level access to hardware sensors. Systems include a four node dual-processor, dual-core AMD 1.8GHz Opteron system running a 2.6.9-11 Linux Kernel, the System X supercomputer (PowerPC G5), and several x86 32- and 64-bit machines with both shared and distributed memory. On all systems, we used GNU C, C++, and/or FORTRAN compilers.

For all experiments (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 ensure 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 to 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.

4.2 Serial Micro-benchmarks

In our first set of experiments, we developed some micro-benchmarks to test the Tempest tool under various conditions. We primarily tested that the sensor data was being traced correctly, that the thermal profiles were as expected, and that overhead was minimal. Table 1 shows results for micro-benchmark D. Figure 2 shows the results in standard output format (part a) and thermal profile output (part b) 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.

Figure 2 Part (a) shows Tempest output divided horizontally into functions (`main`, `foo1`, `foo2`) listed by total execution time (inclusive) spent in each function. The total time heading for each

[3] See http://scape.cs.vt.edu for results for all of the systems, compilers and codes tested and measured.

function provides the amount of time spent in that particular function. Since `main` calls all functions in this case, total time for `main` is the duration of the program. While `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 considered significant for this function. For `foo1` and `main`, all thermal data is shown for each of 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 in Figure 2 part (a).

Figure 2 part (b) plots the temperature trends for each function. Note the y-axis is temperature in degrees Fahrenheit. The x-axis is total execution time in seconds. Also, the duration of each function is show across the top of the figure. We observe `foo1` steadily increases the temperature of the CPU until `foo2` is finally called at which point the temperature drops abruptly while the timer is set and expires. 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.

4.3 Parallel Benchmarks

For brevity, we focus on codes from the NAS parallel benchmark suite. Figures 3 and 4 illustrate the temperature profile data for two of the NAS PB codes.

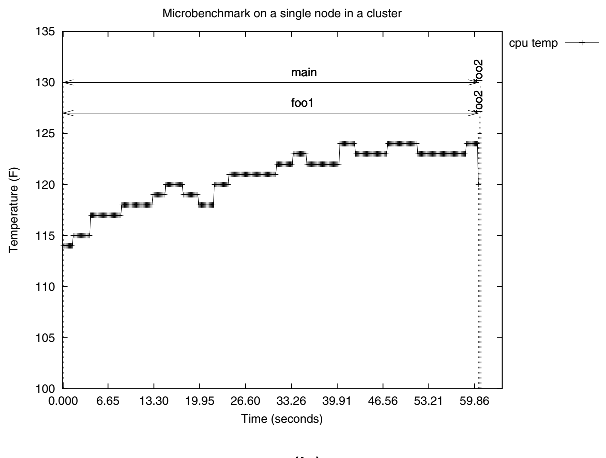
Each graph for a single code is a series of vertically stacked axes with y-axis for temperature in Fahrenheit and x-axis for time in seconds. Each vertical graph for a single code corresponds to nodes in the cluster. The vertical graphs for a single node are vertically aligned so as to aid identification of phase trends in the application. Tables 2 and 3 show the standard output data from Tempest for the same runs on one of the nodes.

The FT benchmark shows very regular behavior in its power profile [3]. Thus, we expected FT (Fourier Transform) which spends 50% of its time in all-to-all communication to run fairly cool. The thermal profiling results were surprising. We observed 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. Clearly power/thermal trends can be quite different despite their inherent relation.

The BT benchmark performs several tasks followed by a synchronization event that occurs at about 1.5 seconds into the run for our class C experiments depicted in Figure 4. This is one of the few codes in the suite that has obviously synchronized thermal characteristics. At the synchronization event, all nodes see a dramatic rise in temperature indicative of increased computation. Surprisingly, some nodes run hotter than others. Nodes 1 and 4 jump above 105 degrees, node 2 stays below, and node 3 runs at over 110 degrees.

```
--------------------------------------------------------------------------------
Function: main        Total Time(sec): 59.860001
--------------------------------------------------------------------------------
             Min      Avg      Max    Sdv    Var     Med      Mod
sensor1  114.00  120.72  124.00  2.73   7.45  121.00  114.00
sensor2   94.00   95.12   97.00  0.56   0.32   95.00   95.00
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Function: foo1        Total Time(sec): 59.828545
--------------------------------------------------------------------------------
             Min      Avg      Max    Sdv    Var     Med      Mod
sensor1  114.00  120.72  124.00  2.73  7.45  121.00  114.00
sensor2  94.000   95.12   97.00   0.56  0.32   95.00   95.00
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Function: foo2        Total Time(sec): 0.000000
--------------------------------------------------------------------------------
             Min      Avg       Max     Sdv     Var     Med     Mod
sensor1  114.00  114.00   114.00   0.00   0.00  114.00  0.00
sensor2   94.00  114.00    94.00   0.00   0.00   94.00  0.00
--------------------------------------------------------------------------------
```

(a)



(b)

Figure 2: Tempest results for micro-benchmark D that is dominated by the foo1 function calling a CPU burn code that heats up the CPU rapidly. Part (a) shows Tempest standard output. Part (b) shows a Tempest temperature profile.

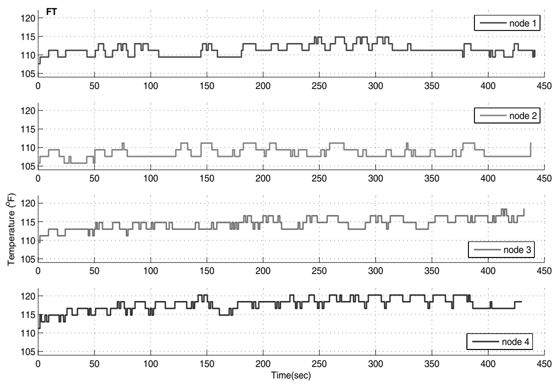


Figure 3: Thermal profile of FT benchmark with NP=4, and Class C.

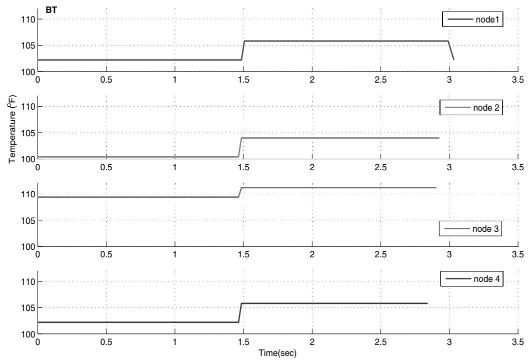


Figure 4: Thermal profile of BT benchmark with NP=4, and Class C.

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 2: Partial Tempest functional profile of FT benchmark with NP=4, 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 3: Partial Tempest functional profile of BT benchmark with NP=4, class C.

5. Conclusions and Future Work

We have designed, implemented, and validated a middleweight tool for fine-grain profiling of the thermal properties of sequential and parallel applications. We used the tool to perform the first fine-grain thermal profiling of parallel scientific applications running on real systems. Our results indicate that 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 the NAS parallel FT and BT benchmarks.

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 level of detail provided by Tempest could identify tradeoffs between various techniques that have not been identified. Finally, we would like to study the use of Tempest data at runtime to make thermal management decisions.

6. References

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

[2] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th annual international symposium on Computer architecture*. Vancouver, British Columbia, Canada: ACM Press, 2000.

[3] K. W. Cameron, R. Ge, and X. Feng, "High-Performance, Power-Aware Distributed Computing for Scientific Applications," *Computer*, 38 (11), pp. 40-47, 2005.

[4] P. Chaparro, J. Gonzalez, and A. Gonzalez, "Thermal-Effective Clustered Microarchitectures," proceedings of First Workshop on Temperature-Aware Computer Systems, Munich Germany, 2004.

[5] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle, "Managing energy and server resources in hosting centers," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*. Banff, Alberta, Canada: ACM Press, 2001.

[6] T. Heath, A. P. Centeno, P. George, L. Ramos, and Y. Jaluria, "Mercury and freon: temperature emulation and management for server systems," *SIGARCH Comput. Archit. News*, 34 (5), pp. 106-116, 2006.

[7] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas, "A framework for dynamic energy efficiency and temperature management," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. Monterey, California, United States: ACM Press, 2000.

[8] W. L. Hung, Y. Xie, N. ViJ'aykrishnan, M. Kandemir, and M. J. Irwin, "Thermal-aware task allocation and scheduling for embedded systems," 2005.

[9] J. Koomey, "Estimating total power consumption by servers in the US and the world," proceedings of EPA Data Centers Technical Workshop, Santa Clara, CA, 2007.

[10] P. Liu, Z. Qi, H. Li, et al., "Fast thermal simulation for architecture level dynamic thermal management," in *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*. San Jose, CA: IEEE Computer Society, 2005.

[11] A. Merkel, F. Bellosa, and A. Weissel, "Event-Driven Thermal Management in SMP Systems," proceedings of Second Workshop on Temperature-Aware Computer Systems, Madison, Wisconsin, 2005.

[12] J. Moore, J. Chase, P. Ranganathan, and R. Sharma, "Making Scheduling "Cool": Temperature-Aware Workload Placement in Data Centers," proceedings of USENIX 2005 Annual Technical Conference, 2005.

[13] K. Skadron, M. R. Stan, K. Sankaranarayanan, et al., "Temperature-aware microarchitecture: Modeling and implementation," *ACM Trans. Archit. Code Optim.*, 1 (1), pp. 94-125, 2004.

[14] K. Skadron, M. R. Stan, H. Wei, et al., "Temperature-aware computer systems: Opportunities and challenges," *Micro, IEEE*, 23 (6), pp. 52-61, 2003.