Using Performance Counters for Runtime Temperature Sensing in High-Performance Processors

Kyeong-Jae Lee and Kevin Skadron
Department of Computer Science
University of Virginia
kl2z@virginia.edu, skadron@cs.virginia.edu

Abstract

As energy consumption in high-performance systems has increased, thermal management has become a big challenge. Providing a cost-effective and detailed temperature sensing mechanism is crucial to effectively employ a thermal management technique. Existing hardware sensors are too costly to implement and add additional heat while software simulations fail to account for all possible hardware effects.

In this paper, we describe a software solution for temperature sensing that uses real hardware resources such as performance counters. The resulting temperature model provides a detailed spatial gradient of the processor and executes at runtime. In particular, the model is configured for the Pentium 4 processor. We run SPEC2000 benchmarks to analyze the thermal behavior of applications and explain the potential benefits of using our model for temperature-aware research.

1. Introduction

The main objective of this paper is to present a runtime temperature sensing methodology that can be used for high-performance architecture studies on real hardware, even in the absence of fine-grained on-chip sensor hardware. We then use this methodology to show how the temperature model can be used to characterize high-performance applications and guide temperature-aware design. This will be achieved by analyzing SPEC benchmark results on a Pentium 4 processor.

High performance drives development of many categories of processors. As a result, power and heat density of processors has increased exponentially with Moore's Law to the point that they affect system performance, accelerate aging of the chip, and increase the cost of cooling solutions. As an effort to reduce heat dissipation, researchers have developed various dynamic thermal management (DTM) techniques, such as clock gating and dynamic voltage scaling [2, 3, 4, 9]. DTM techniques monitor and control the temperature of the chip at runtime.

While temperature-aware design has gained importance in the field, most control mechanisms require the ability to accurately measure the temperature of the processor; the DTM detects when the temperature reaches a certain threshold and then regulates processor operations. Temperature sensing methods are also important for architecture studies, to allow researchers to examine thermal effects or test DTM algorithms. Skadron et al proposed a thermal model called *HotSpot*, which computes the temperature for each microarchitecture block on the chip [10]. Their thermal model can interface with power-performance simulators for architecture studies. Simulation is an inexpensive way to get results, but fails to account for all system-wide hardware effects and can be prohibitively slow. Obtaining real temperature readings directly from the chip's temperature sensors would be ideal, but there are several limitations to this. First, most of the sensors are based on analog CMOS circuit designs, and can be costly to implement and may even exacerbate the thermal problem by dissipating too much power. Second, a chip usually contains only a few numbers of sensors so their placement on the chip becomes an important design issue. Third, the sensor's response time to a temperature change can be too slow unless it is implemented in CMOS in the chip. Hence, an accurate overall representation of the thermal distribution of the chip at runtime can be difficult to obtain through thermal sensors.

This paper proposes a new runtime thermal model and temperature sensing architecture that extends HotSpot to interface with physical hardware resources such as performance counters. Performance counters can be configured to gather specific micro-architectural events such as cache hits, and are therefore a good estimate of processor activity. Isci and Martonosi have already shown that accurate runtime modeling of power is possible using performance counters [8]. Our proposed thermal model attempts to replace the use of thermal sensors and provide a detailed, run-

time, floorplan-level profile of the chip's temperature distribution. We focus on the Intel Pentium 4 processor and its architecture.

The paper is structured as follows. Section 2 gives an overview of the requirements and implementation details. Section 3 provides an analysis of the model in terms of performance penalty and current limitations. Section 4 presents several thermal maps of the Pentium 4 as an example how researchers could use the model to improve hardware design in addition to its use for temperature sensing. Section 5 concludes the paper.

2. Implementation

2.1. Pentium 4 Architecture

The machine used for the experiments is a 2.6 GHz Pentium 4 processor, 130 nm Northwood core. The typical power dissipation is 69.0 W, and the operating voltage is 1.6 V [7]. The Pentium 4 uses a 20-stage pipeline and a trace cache, which eliminates the normal instruction decoding from the execution loop by storing IA-32 instructions. The two ALUs each execute in one-half the clock cycle, and thus double the effective throughput. The machine also supports hyper-threading technology, which allows the processor to run two threads simultaneously. Because the temperature model is constantly running in the background, hyper-threading allows calculations from the model to run concurrently with other programs with less overhead. The Pentium 4 also includes a rich set of performance monitoring features, with 45 configurable events and 18 physical performance counters [5, 12]. The performance counters are used to count specific micro-architectural events for debugging and performance measurements. Each counter is associated with one counter configuration control register (CCCR), which determines the specific counting scheme. The event selection control registers (ESCRs) determine which event is to be counted. A simplified device driver, adapted from the abyss device driver [11], is used to configure all the control registers and read the performance counter values.

The temperature model also requires the geometric specifications and the floorplan layout of the processor. Table 1 shows the mechanical dimensions and material characteristics for components in the processor package. These settings are based off of design schematics found in [7] and are used to configure the HotSpot simulator program. Figure 1 is an approximated floorplan layout that has been adapted from the Northwood core die photo [6]. It includes the following functional units: L1 branch prediction unit (BPU), L2 BPU, instruction decoder, trace cache, memory order buffer (MOB), ITLB, bus control unit, DTLB, L1 cache, L2 cache, micro-code ROM (UROM), allocation unit, rename unit, instruction Q1, instruction Q2, scheduler, retirement unit, FP execution unit, FP register file, integer execution unit, integer register file, and the memory control unit. The trace cache and L1 cache have been divided into two units for simplicity.

Table 1. HotSpot simulator configuration

HotSpot variable	Value	Description (Unit)
t_chip	0.00074	chip thickness (m)
therm_threshold	135+273.15	DTM threshold (K)
c_convec	295.7	convection capacitance (J/K)
r_convec	0.467	convection resistance (K/W)
s_sink	0.076	heat sink side (m)
t_sink	0.0411	heat sink thickness (m)
s_spreader	0.031	heat spreader side (m)
t_spreader	0.0015	heat spreader thickness (m)
t_interface	0.000127	interface material thickness (m)
ambient	45+273.15	ambient temp (K)

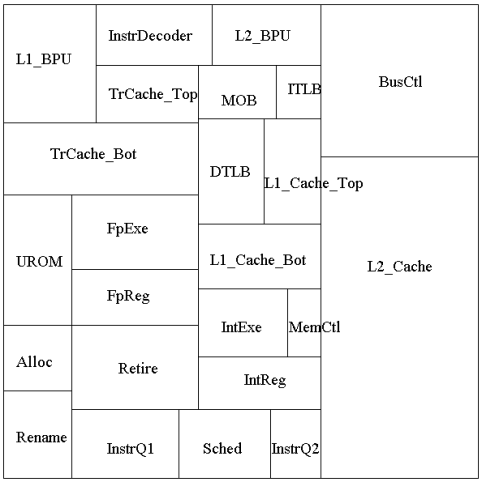

Figure 1. Floorplan layout of Pentium 4 (adapted from chip-architect.com)

2.2. HotSpot Extension and Power Modeling

The proposed model extends the HotSpot thermal model developed by Skadron et al [10]. HotSpot has already been validated against a thermal test chip and IBM finite-element simulations. Currently, HotSpot models the processor as a network of thermal resistors and conductors per functional

unit, with power dissipation in each unit treated as a current source in the RC network. The proposed model needs to estimate power dissipation from performance counters. Isci and Martonosi have already shown that power can be accurately modeled from performance counters [8]. Their model uses the following equation:

$$\begin{aligned} Power &= AccessRate \\ &\times ArchitecturalScaling \\ &\times MaxPower \\ &+ NonGatedClockPower \end{aligned} \quad (1)$$

Several micro-architectural events, which are measured through performance counters, are combined to closely approximate the number of accesses to each functional unit. We use similar metrics found in [8] and extend HotSpot to interface with performance counters. For the Pentium 4, not all performance metrics can be measured simultaneously using the 18 performance counters. Four sets of counter rotations are required to sample all necessary architectural events. Thus, the performance counters are periodically sampled but a different set of architectural events is measured each time. Although only two counters are time-shared by different metrics across all four rotations, some events will be missed and the temperature values may eventually need to be calibrated against the hardware sensor. Refining the model to further improve its accuracy is an area for future work.

In addition, the main temperature computation algorithm must be optimized to satisfy the runtime requirements. For example, if we are updating the temperature values every 10 ms, the actual sampling and calculation performed by the program must be less than 10 ms. This also assumes that the program is multithreaded. Currently, HotSpot uses a 4th order Runge-Kutta solution to calculate temperature. This solution was designed with a conservative step size and proved to be inadequate for runtime measurements despite some parameter changes. In our model, we implement the Runge-Kutta-Fehlberg method, which uses an adaptive step size to minimize the calculation time. Despite the complexity, the RKF method is more efficient and can be easily integrated into the existing HotSpot framework. Further optimizations of the HotSpot model are an important area for future work.

3. Results

3.1. Program Overhead

The temperature model was compiled using gcc-3.2.2 with compiler flags of "-O3 -march=pentium4 -mfpmath=sse -mmmx -msse -msse2". All benchmark programs were run on the linux kernel 2.4.20-24.9smp. When the temperature model executes, it periodically prints out a list of temperature values for each functional unit. We use the default calling interval for all experiments; 5 ms for each counter-rotation, and 20 ms to update temperature values. Although the temperature model updates temperature values infrequently, the program continually monitors access to performance counters and updates power values. It also needs to perform a large set of calculations to obtain temperature values; these all add to the system overhead. Two metrics are used to find the inherent overhead of the program.

Table 2. Thermal Overhead (^{o}C)

Unit	Steady-state	Two Instances	Overhead
BusCtl	47.73	51.24	3.51
L2_Cache	50.08	54.19	4.11
L2_BPU	56.11	64.02	7.91
InstrDecoder	55.12	61.24	6.12
L1_BPU	55.71	62.04	6.33
ITLB	51.57	57.03	5.46
TrCache_Top	58.00	63.79	5.79
TrCache_Bot	58.26	63.50	5.24
DTLB	56.65	63.33	6.68
L1_Cache_Top	53.96	60.37	6.41
L1_Cache_Bot	58.79	66.93	8.14
IntExe	66.33	79.18	12.85
MemCtl	64.43	73.35	8.92
IntReg	70.03	83.68	13.65
FpExe	55.65	59.27	3.62
FpReg	59.31	63.30	3.99
UROM	52.63	56.13	3.50
Alloc	74.83	79.81	4.98
Rename	76.24	81.13	4.89
Retire	69.76	75.59	5.83
InstrQ1	76.59	82.22	5.63
Sched	74.05	82.25	8.20
InstrQ2	69.48	77.69	8.21

To estimate the thermal overhead, we obtain the steady-state temperature values for each functional unit. We run the model by itself until the temperature reaches steady state. Then we execute two slightly different instantiations of the model simultaneously. Since there is no natural way of knowing the temperature per block without using the model, the amount of heat generated by the model itself is approximated as the difference between the values from this experiment and the steady-state condition. One instantiation is the normal program while the other is modified so that counter values are statically assigned instead of obtaining real counter values via the device driver. This approximation serves as a general indicator since the major-

ity of the CPU activity is in the HotSpot calculation algorithm and not in the performance monitoring functionality. Table 2 shows that for most units the increase in temperature is around 4 to 8 ^{o}C.

To estimate the performance overhead, we obtain the execution time of SPEC benchmarks. All SPEC benchmarks were compiled using the SPEC "base" tuning option. Figure 2 shows the execution time of benchmarks while running the temperature model simultaneously. All values are normalized to the execution time of each benchmark running alone. The overhead of the execution time varies across applications, and the variation is larger across floating-point benchmarks than integer benchmarks. The temperature model's main computation algorithm requires several iterations of the Runge-Kutta algorithm. Benchmarks with iterative numerical methods that stress floating-point units are more likely to compete in hyper-threading with the temperature model for CPU resources, and hence the overhead is greater for these benchmarks. Benchmarks that have high overhead are of special interest since those programs produce more thermal stress and can be more easily used to characterize their thermal behavior. Section 4 includes a more detailed discussion of the thermal characteristics of the gzip and wupwise benchmarks.

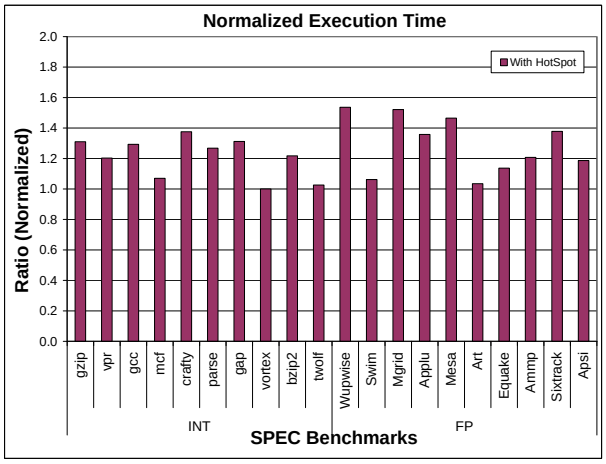

Figure 2. Performance overhead

3.2. Limitations

The temperature model processes performance statistics and outputs a temperature reading for each functional block of the Pentium 4. This temperature sensing methodology provides three main advantages over other sensing mechanisms: the model provides a detailed spatial thermal distribution, calculation is based on real hardware activity, and the information is presented at runtime. The model facilitates architecture studies that can help guide temperature-aware design and chip floorplanning for high-performance computing. Section 4 explains how this model can be used to analyze thermal stress patterns and understand how such patterns are dependent on the application characteristics. However, as evident from Section 3.1, the high overhead of this program becomes problematic for high-performance applications that need greater precision, faster response times, and lower runtime costs. This may not be as critical for architectural studies since researchers can quantify the overhead. Although the calculations are required infrequently, multithreading the performance counter sampling and temperature calculations can overload the processor. The high increase of temperature in the integer units in Table 2 also show that the proposed model is more likely to slow down computation-intensive applications.

Future research may explore ways to reduce the overhead of the model. One method is through optimizing the software code. The heavy computation needs to be reduced either by improving upon the Runge-Kutta numerical solution or by replacing it with a more efficient algorithm. It may even be desirable to trade off some accuracy to obtain a radically simpler algorithm. Bellosa et al also use performance counters as an event-driven approach, but calculate a single temperature value by solving differential equations [1]. Compiler optimization may also help. For example, when running the wupwise benchmark, the use of machine-specific compiler flags (e.g., -march=p4, -mmx, etc) for the temperature model reduced the thermal stress on the chip. While the effect is minimal for most functional units, the integer execution unit and register file showed a temperature reduction of 4.1 ^{o}C. An alternative to optimizing the source code is to exploit various hardware resources. One method would be to generate a sensor-fusion algorithm where readings from the thermal sensor and software program are combined (see Section 4.3). It would be possible to create an algorithm to infer the thermal distribution from a sensor and invoke the program only when needed. This can minimize excess computation and still take advantage of the existing sensors.

4. A Case Study of the Pentium 4

4.1. Thermal Stress Patterns

The proposed temperature model is a cost-efficient software solution that can provide reasonable estimates of temperature. This information is useful for studying thermal behavior of high-performance applications and temperature-aware designs. In this section, we use thermal maps of the Pentium 4 to illustrate the thermal characteristics of the SPEC benchmarks. While the information is applicable to most of the benchmarks, we focus on the gzip inte-

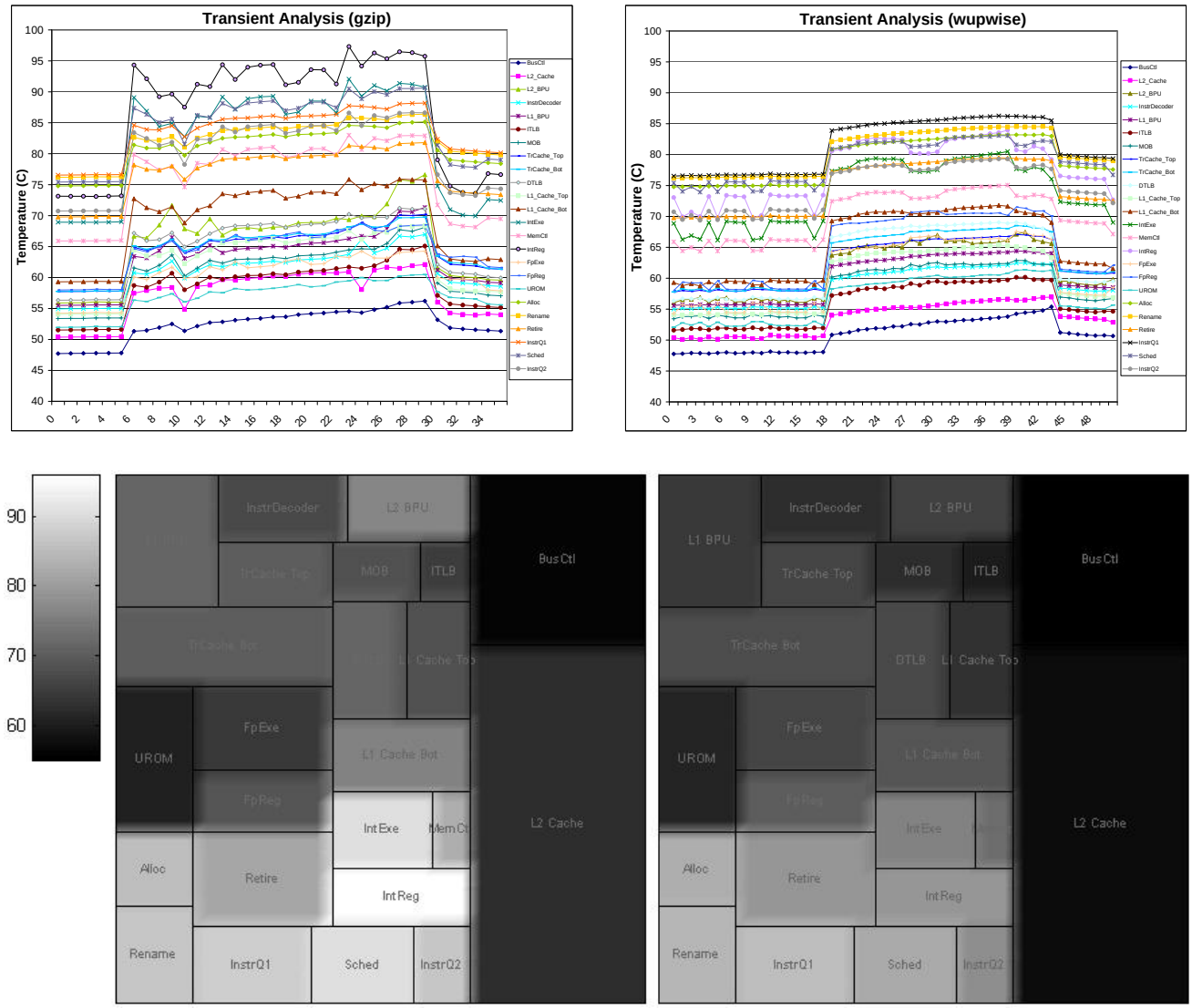

Figure 3. (a) Temperature change over time (gzip) (b) Temperature change over time (wupwise) and (c) Spatial difference for gzip and wupwise benchmarks

ger benchmark and wupwise floating-point benchmark. Table 2 in Section 3.1 shows that the allocation, rename, instruction Q, scheduling units are generally the hottest units on the chip. This seems reasonable, since every instruction must be processed through the pipeline. Figure 3 shows the change in temperature over time for the gzip and wupwise benchmark. For integer benchmarks, the integer units are typically the hottest units on the chip. In comparison to the steady-state condition, the amount of increase in temperature for each functional block during peak execution varies from 4 to 25 ^{o}C. In contrast, the floating-point benchmarks show a relatively uniform temperature increase over all functional blocks; the range of the amount of temperature increase, 6 to 12 ^{o}C, is smaller than that of the integer benchmarks. While the floating-point units still heat up the most for floating-point benchmarks, the overall thermal distribution does not differ much from the steady-state condition other than the fact that the average temperature is higher. Figure 3 also shows a thermal plot of the processor for each benchmark, gzip and wupwise, during its peak execution. The larger temperature range across the entire processor for the gzip benchmark is easily noticeable.

In addition to the application-specific spatial differences, our model can be used to demonstrate how hot spots can move during the execution of an application. Figure 4 shows a temperature trace of the gcc integer benchmark.

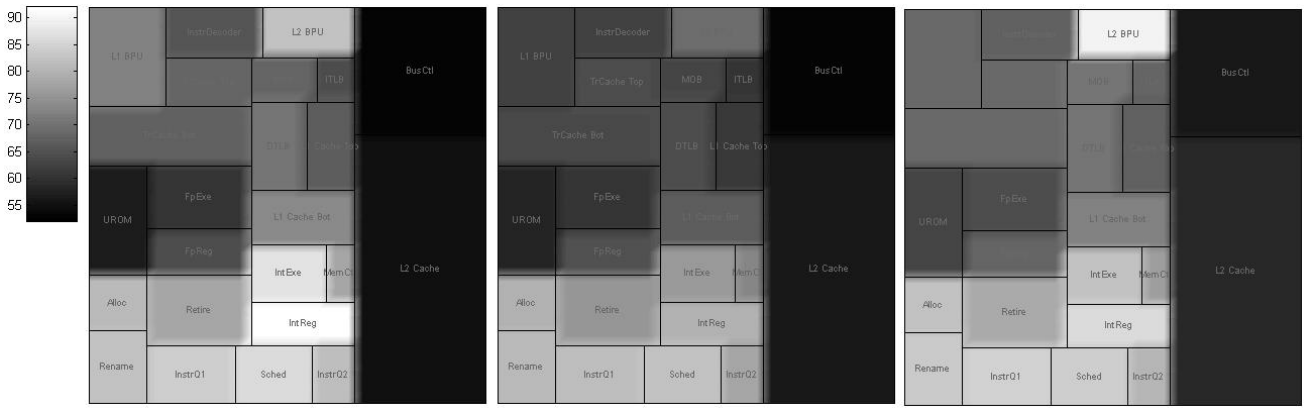

Figure 5. Hot spot movement. Hottest block moves from the integer register file to the scheduler unit, and then to the L2 BPU

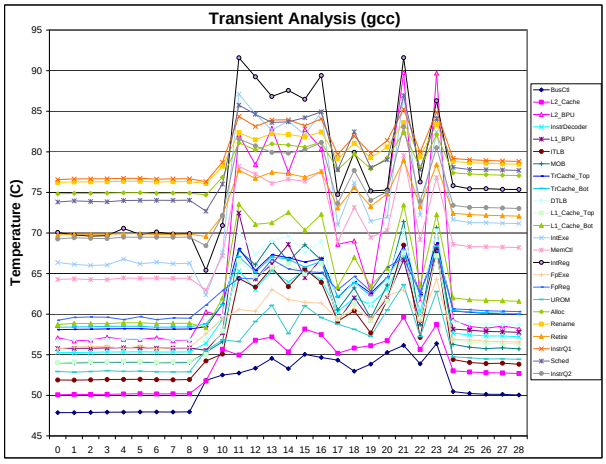

Figure 4. Temperature change over time (gcc)

Note that the maximum temperature value does not come from the same functional unit during execution. This is more clearly illustrated in Figure 5. The three thermal plots in Figure 5 are separate instances in time during the execution of the gcc benchmark. The hottest block moves from the integer register file to the scheduler unit, and then to the L2 BPU.

4.2. Hardware Design Improvements

This section includes a discussion how the thermal mapping of the chip can guide processor designers to develop thermal-efficient systems.

Our model can provide useful information when manufacturers are designing the processor package. Results from Section 4.1 suggest that floating-point benchmarks tend to uniformly stress the processor with less temperature variation across functional units than integer benchmarks. This indicates that it may be feasible to use less expensive packaging materials for computer systems that chiefly support numerical computations. Since integer benchmarks tend to create large temperature gradients around the integer units, differentiating the heat spreader material around the known hot spots would be a possible solution to effectively remove heat for processors running integer applications.

In addition, our temperature model can help designers identify the thermal gradient bounds and intelligently place thermal sensors. Gunther et al have explained the importance of examining thermal maps to find the optimal location of sensors on the Pentium 4 [4]. While they used a simulated thermal plot, our temperature model presents a more realistic representation of temperature through the use of performance counters. Hence, hot spots are more accurately identified through our model.

The Pentium 4 has two thermal sensors, and one acts as a catastrophic shutdown detector. An interesting question to consider is whether hot spots could be artificially induced where the sensor cannot effectively detect them. Complete or partial meltdown of the processor can raise enormous thermal security risks. Even unexpected shutdowns cause an availability and possible data integrity risk. Consider the thermal maps in Figure 3 in Section 4.1. Note that the hottest block for the wupwise benchmark is the instrQ1 unit, which is further away from the coolest unit (i.e., the BusCtl unit) than the integer register file is for the gzip benchmark. The gzip benchmark stresses integer units and creates a larger gradient between the coolest and hottest area (40 ^{o}C) than the wupwise benchmark (30 ^{o}C). If a program can target a specific functional unit to overheat more quickly than the thermal control circuit can respond, it may be possible that the throttling is not engaged fast enough. This

illustrates another potential use of the temperature model: to study thermal viruses that can cause unexpected system shutdown.

4.3. Hybrid Hardware-Software Solution for Thermal Sensing

One promising area for future research is to combine our temperature sensing approach with existing hardware sensors. A hybrid hardware-software solution may prove to be the most cost-effective method to obtain an accurate and detailed temperature sensing method that can be used for DTM. It would be possible to create a model that uses performance counters as a proxy for temperature. The new model could continually monitor the counter values, and re-caclulate the temperature for each functional unit only when the gradient is large enough or when the temperature is reaching a certain threshold. We can use the thermal sensors of the Pentium 4 to callibrate the new temperature values based on the floorplan layout and the location of the sensor.

Compared to a purely hardware-based or software-based solution, this sensor-fusion algorithm can reduce the overall cost and provide the benefits of both approaches. Receiving input from the hardware sensors effectively minimizes the number of times the temperature values are calcuated, and hence the computation overhead of the software (see Section 3) can be reduced. In addition, the system can obtain a detailed spatial distribution of temperature through software modeling even in the absence of fine-grained on-chip sensors.

5. Conclusions

This paper has presented a runtime thermal model configured for the Pentium 4 processor. This software solution provides detailed temperature information at the floorplan level and uses the hardware performance counters as a measure of real processor activity. The software can be used for both architectural studies and on-line temperature sensing. We have shown early work of the potential benefits, but a great deal of future work is required for our approach to fully develop as a low-cost and reliable temperature sensing mechanism. The current implementation of the model adds about 4 to 8 ^{o}C of thermal overhead and can slow down the execution time of SPEC benchmarks by up to 54 %. The main bottleneck of the program is in the multithreaded code for performance monitoring and temperature calculations. If the model is further optimized, it can then be used for high-performance applications that require low runtime costs, or even as a cost-effective alternative to thermal sensors for runtime DTM techniques.

In addition, the Pentium 4 has only two thermal sensors and hence the accuracy of our current temperature model has not yet been fully validated. One of the two sensors is accessible through software, and the temperature estimate from our model has been shown to closely match the reading from the hardware sensor. Furthermore, the relative heatup and stress patterns of the processor still provide useful information for temperature-aware architecture studies as shown in Section 4. One possible extension is to apply our sensing methodology to IBM's Power5 processor, which has 24 thermal sensors. Not only will it be possible to validate our sensing methodology, but it will also allow researchers to study a hyrbid hardware-software temperature sensing solution that may be more efficient than a hardware solution of many sensors.

We also showed how the model can be used for thermal-aware, high-performance computing studies. We used the gzip and wupwise SPEC2000 benchmarks as an example to highlight some of the key thermal features of the integer and floating-point benchmarks. Our results show that integer benchmarks tend to have more variation than floating-point benchmarks in terms of temperature changes across functional units. Localized hot spots can also move across the processor during program execution. Understanding application-specific thermal behavior can guide researchers in designing new thermal-efficient processor packages, floorplan layouts, and in placing thermal sensors on the chip. We hope our sensing approach will be used as a tool for future studies on thermal effects in high-performance computing.

Acknowledgments

This work is supported in part by the National Science Foundation under grant no. CCR-0133634, a grant from Intel MRL, the University of Virginia Fund for Excellence in Science and Technology, and the University of Virginia Summer Science and Engineering Scholars Program. We would also like to thank Mircea Stan, Karthik Sankaranarayanan, and Puyan Dadvar for their constructive feedback.

References

[1] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner. Event-driven energy accounting for dynamic thermal management. In *Proc. COLP 2003*, Sep. 2003

[2] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proc. HPCA-7*, pp.171-82, Jan. 2001

[3] M. Fleischmann. Crusoe power management: Cutting x86 operating power through LongRun. In *Embedded Processor Forum*, June 2000

[4] S. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Tech. J.*, Q1 2001

[5] Intel Corporation. IA-32 Intel Arch. Software Developers Manual, Vol. 3: System Programming Guide, 2004

[6] Intel Pentium 4 Northwood die photo. From website: Chip Architect. http://www.chip-architect.com/news/2003_04_20_Looking_at_Intels_Prescott_part2.html

[7] Intel Pentium 4 technical documents. From website: Intel Corporation. http://www.intel.com/design/Pentium4/documentation.htm

[8] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proc. MICRO-36*, Dec. 2003

[9] M. Ma et al. Enhanced thermal management for future processors. *VLSI Circuits*, pp.201-204, Jun. 2004

[10] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proc. ISCA-30*, Apr. 2003

[11] B. Sprunt. Brink and Abyss Pentium 4 Performance Counter Tools For Linux, Feb. 2002. http://www.eg.bucknell.edu/bsprunt/emon/brink_abyss

[12] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):7282, Jul/Aug 2002