Temptor: A Lightweight Runtime Temperature Monitoring Tool Using Performance Counters

Yongkui Han, Israel Koren and C. M. Krishna
Department of Electrical and Computer Engineering
University of Massachusetts,Amherst, MA 01003
E-mail: {yhan,koren,krishna}@ecs.umass.edu

ABSTRACT

Temperature projecting is a crucial component of temperature aware techniques since most of them rely on either CPU temperature estimations or measurements. In this paper, we first present a new transient thermal simulation algorithm, TILTS, which is much faster than conventional simulation algorithms. Based on the TILTS algorithm, we propose a lightweight runtime temperature monitoring tool called *Temptor*. Using internal performance counters, *Temptor* is able to estimate runtime temperature distributions in CPU chips with a negligible overhead. *Temptor* provides detailed spatial temperature distribution information in CPU chips, enabling us to study various thermal issues in high performance microprocessors. Experimental results illustrating the use of *Temptor* are presented.

1. INTRODUCTION

Power density has been increasing in each generation of microprocessors since feature size and frequency are scaling faster than the operating voltage. Power density directly translates into heat, and consequently processors are getting hotter. For example, Pentium 4 chips generate more heat per surface area than a kitchen hotplate and Intel's projections show that the heat per surface area generated by its processors will increase sharply in the coming years, approaching that of the core of a nuclear power plant, unless solutions to this problem can be found [9].

Removing the excessive heat is also a challenge. In order to keep the chip temperature below a certain limit, the heat generated by the processor must be rapidly dissipated. As a result, the cost of removing heat is increasing at about the same rate as power density and the cost of the cooling system constitutes a major component of the overall cost of the computer system.

In order to study temperature issues in processors, Skadron et al. developed the HotSpot software tool [1][17], which calculates the temperature distribution among different blocks in the CPU chip. The HotSpot thermal model is a dynamic compact thermal model based on the well-known similarity between the flow of heat and that of electric currents: both are described by similar differential equations. Heat flow can be described as a current passing through a thermal "resistor", leading to a temperature difference analogous to a voltage. Lumped values of thermal resistance and capacitance can be computed to represent the heat flow among units. Based on this duality, equivalent circuits called compact RC models can be constructed [26]. HotSpot is an architectural level thermal simulator that can be used together with other architectural level simulators, such as SimpleScalar [13] or Wattch [11].

In an effort to reduce heat generation, researchers have developed various dynamic thermal management (DTM) techniques such as clock gating, dynamic voltage scaling, and dynamic frequency scaling [10][14][25][15][26][27]. DTM techniques monitor and control the temperature of the chip at runtime. Most DTM techniques rely on either measured or estimated temperatures.

Taking into consideration the internal performance monitoring capabilities of contemporary high-end microprocessors, Isci and Martonosi [18] describe a coordinated measurement approach that combines real total power measurements with performance-counter-based, per-unit power estimations. The resulting tool offers real-time total power measurements for Intel Pentium 4 processors, and also provides power breakdowns for 22 major CPU components over several minutes of SPEC2000 and desktop benchmarks. The generated component power breakdowns are used to identify program power phase behavior in [18, 20]. The tool has not been used for temperature aware techniques.

In [22], Lee and Skadron describe their software solution for temperature sensing that uses physical performance counters inside the CPU. Their resulting temperature model provides a detailed spatial gradient of the processor and executes at runtime. Because of the high overhead of HotSpot temperature calculations, the performance of the original benchmark application is affected significantly. To reduce this overhead, instead of using the original 4th order Runge-Kutta method (*rk4*) in Hotspot, they have implemented an improved Runge-Kutta-Fehlberg method (*rkf*) with adaptive step size to minimize the temperature calculation time. The *rkf* method is more efficient than the *rk4* method despite its complexity. Even with the improved *rkf* method, they have observed more than 50% performance degradation for some SPEC2000 benchmarks.

In this paper, an approach similar to that in [22] is adopted, but the bottleneck in the efficiency of temperature calculations is removed. We first propose a new transient thermal simulation algorithm **TILTS** (**T**ime **I**nvariant **L**inear **T**hermal **S**ystem), which is much faster than conventional simulation algorithms. Then, based on the TILTS algorithm, we develop a lightweight runtime temperature monitoring tool, *Temptor* (*Temp*erature Moni*tor*). Using the internal performance counters, *Temptor* is able to estimate runtime temperature distributions in CPU chips.

In summary, our contributions are as follows:

1. We present a new transient thermal simulation algorithm (called TILTS) that is much faster than conventional simulation algorithms without any loss in accuracy.

2. Based on the TILTS algorithm, we develop a lightweight runtime temperature monitoring tool (called *Temptor*). Using performance counters inside the Pentium 4 CPU, the software tool *Temptor* is able to estimate the runtime temperature distribution of the CPU chip.

3. Using *Temptor*, we study the thermal behavior of SPEC2000 benchmarks.

The rest of the paper is organized as follows. In Section 2, we

present our fast transient thermal simulation algorithm. In Section 3, we propose a framework for a lightweight runtime temperature monitoring tool and describe the implementation based on the *perfctr* device driver. The experimental results for the Pentium 4 microprocessor are given in Section 4. Conclusions and acknowledgements are presented in Sections 5 and 6, respectively.

2. A FAST TRANSIENT THERMAL SIMULATION ALGORITHM

2.1 Linear System Theory Overview

TILTS uses linear system theory to model chip temperatures. A linear system [23] is described by its state equation, where the state variables are system internal variables. Denote the number of state variables by N, the number of inputs to the system by M, and the state and input vectors by $\mathbf{x}(t)$ and $\mathbf{u}(t)$, respectively:

$$\mathbf{x}(t) = (x_1(t), x_2(t), ..., x_N(t))^T$$

$$\mathbf{u}(t) = (u_1(t), u_2(t), ..., u_M(t))^T$$

The linear system equation is:

$$\dot{\mathbf{x}}(t) = \mathbf{F}\mathbf{x}(t) + \mathbf{G}\mathbf{u}(t) \tag{1}$$

where $\mathbf{F}$ is an $N \times N$ matrix, and $\mathbf{G}$ is an $N \times M$ matrix. These matrices are constant for a time-invariant linear system.

For such a linear system, the complete response is the sum of the zero-input response and the zero-state response.

$$\mathbf{x}(t) = e^{\mathbf{F}t}x(0) + \int_0^t e^{\mathbf{F}(t-\tau)}\mathbf{G}\mathbf{u}(\tau)d\tau \tag{2}$$

The first term on the right hand side is the zero input response due to the initial condition and the second term is the response to the input impulse, $\mathbf{u}(t)$.

2.2 CPU Chip as a Linear System

A CPU chip is a thermal system that can be described by its equivalent thermal circuit composed of thermal resistors and capacitors. All these components are linear components, making it a linear system. The input to this linear system is the power dissipated by each functional unit on the chip, and its state variables are the temperatures of the internal nodes in the thermal circuit.

Let M be the number of functional units which dissipate power, and N the number of internal nodes in the thermal circuit ($N \geqslant M$). Denote the thermal resistance between node i and j by r_{ij}, and the thermal capacitance to the thermal ground (the ambient environment) by c_i for node i. For convenience, let $1/r_{ii} = 0$ in the following summation equation. The CPU thermal system obeys the following differential equation:

$$c_i\dot{x}_i(t) = -\sum_{j=1}^{N}\frac{1}{r_{ij}}(x_i(t) - x_j(t)) + \tilde{u}_i(t), i = 1, 2, ..., N \tag{3}$$

where $\mathbf{x}(t) = (x_1(t), ..., x_N(t))^T$ is the temperature vector and $\tilde{\mathbf{u}}(t) = (u_1(t), ..., u_M(t), 0, ..., 0)^T$, i.e., $\tilde{\mathbf{u}}(t)$ is $\mathbf{u}(t)$ (the power vector) extended by $N - M$ zeros, corresponding to nodes which have no power dissipation associated with them (e.g., thermal interface material, heat spreaders, heat sinks, etc.).

Let $\mathbf{D} = (d_{ij})_{N\times N}$, $\mathbf{C} = (c_{ij})_{N\times N}$, where

$$d_{ij} = \begin{cases} -\sum_{k=1}^{N}\frac{1}{r_{ik}} & \text{if } i = j, \\ \frac{1}{r_{ij}} & \text{if } i \neq j. \end{cases} \tag{4}$$

$$c_{ij} = \begin{cases} c_i & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases} \tag{5}$$

Then equation (3) can be rewritten as:

$$\mathbf{C}\dot{\mathbf{x}}(t) = \mathbf{D}\mathbf{x}(t) + \tilde{\mathbf{u}}(t) \tag{6}$$

$\mathbf{C}$ is a diagonal matrix and thus it is easy to compute its inverse $\mathbf{C}^{-1}$ and obtain the standard differential equation:

$$\dot{\mathbf{x}}(t) = \mathbf{C}^{-1}\mathbf{D}\mathbf{x}(t) + \mathbf{C}^{-1}\tilde{\mathbf{u}}(t) \tag{7}$$

Note that since $\tilde{u}_i = 0$ for $i > M$, only the left M columns of $\mathbf{C}^{-1}$ are useful in the second term in (7). We can therefore, construct an $N \times M$ matrix G out of the left M columns of $\mathbf{C}^{-1}$ and replace $\tilde{\mathbf{u}}$ by $\mathbf{u}$. Thus, we obtain an equation similar to (1) with

$$\mathbf{F} = \mathbf{C}^{-1}\mathbf{D} \text{ and } \mathbf{G} = \text{left } M \text{ columns of } \mathbf{C}^{-1} \tag{8}$$

Therefore, a formula similar to (2) can be used to calculate the transient temperature of the CPU.

The input power trace to the CPU is usually given as a series of power vectors. In a sampling interval Δt, the power vector $\mathbf{u}(t)$ is constant allowing us to simplify equation (2) as follows:

$$\mathbf{x}(\Delta t) = e^{\mathbf{F}\Delta t}\mathbf{x}(0) + \Big[\int_0^{\Delta t} e^{\mathbf{F}(\Delta t-\tau)}\mathbf{G}d\tau\Big]\cdot\mathbf{u} \tag{9}$$

We use this simplified equation to reduce the amount of computation. Denoting

$$\mathbf{A} = e^{\mathbf{F}\Delta t}, \quad \mathbf{B} = \int_0^{\Delta t} e^{\mathbf{F}(\Delta t-\tau)}\mathbf{G}d\tau \tag{10}$$

we obtain the equation:

$$\mathbf{x}(\Delta t) = \mathbf{A}\mathbf{x}(0) + \mathbf{B}\mathbf{u} \tag{11}$$

Because the system is a time-invariant linear system, we obtain the same equation for any interval Δt with the same matrices $\mathbf{A}$ and $\mathbf{B}$:

$$\mathbf{x}(n\Delta t) = \mathbf{A}\mathbf{x}((n-1)\Delta t) + \mathbf{B}\mathbf{u}(n-1) \tag{12}$$

where $\mathbf{u}(n-1)$ is the power vector in the time interval $[(n-1)\Delta t, n\Delta t]$.

We will use $\mathbf{x}(n)$ to represent $\mathbf{x}(n\Delta t)$ for conciseness, resulting in:

$$\mathbf{x}(n) = \mathbf{A}\mathbf{x}(n-1) + \mathbf{B}\mathbf{u}(n-1) \tag{13}$$

2.3 Time Invariant Linear Thermal System (TILTS) Algorithm

Our transient thermal simulation method TILTS is based on equation (13). Suppose the number of power vectors (called data points) in the input power trace $\mathbf{u}$ is n, and the initial temperature is $\mathbf{x}_0$. The algorithm TILTS is shown below:

Algorithm TILTS($\mathbf{x}_0, \mathbf{u}, n$)

1. Calculate matrices $\mathbf{D}$ and $\mathbf{C}$ using HotSpot. Then calculate matrices $\mathbf{F}$ and $\mathbf{G}$ using (8). Finally, calculate matrices $\mathbf{A}$ and $\mathbf{B}$ for the sampling interval Δt using (10).
2. For $i = 1$ to n do $\mathbf{x}(i) = \mathbf{A}\mathbf{x}(i-1) + \mathbf{B}\mathbf{u}(i-1)$.

Denoting $\mathbf{A} = [\mathbf{a}_1\mathbf{a}_2...\mathbf{a}_n], \mathbf{B} = [\mathbf{b}_1\mathbf{b}_2...\mathbf{b}_m]$, where $\mathbf{a}_i, i = 1, ..., N$ and $\mathbf{b}_i, i = 1, ...M$ are the i-th column vectors in $\mathbf{A}$ and $\mathbf{B}$, respectively, then for a given time interval $[0, \Delta t]$,

$$\mathbf{x}(\Delta t) = \begin{cases} \mathbf{a}_i, & \text{if } x_i = 1, x_j = 0 (j \neq i), u_j = 0, \\ \mathbf{b}_i, & \text{if } u_i = 1, u_j = 0 (j \neq i), x_j = 0. \end{cases} \tag{14}$$

Table 1: Number of iterations of *rk4* in the HotSpot simulator.

sampling interval Δt	# of *rk4* iterations in HotSpot
3.33 μs	9
5 μs	13
10 μs	26
20 μs	51
50 μs	128
5 ms	12800
40 ms	102400

That is, each column vector of the matrices A and B is the step response to either one x_i or one u_i only. In our algorithm implementation, HotSpot is used to calculate the step response to a single x_i or u_i over time Δt, and then the matrices $\mathbf{A}$ and $\mathbf{B}$ are obtained using (14). Therefore, while the calculation of the matrices $\mathbf{A}$ and $\mathbf{B}$ still uses the conventional integration-based method, this calculation is only performed once with a computation time which is less than 0.01 seconds.

2.4 Performance of the TILTS Algorithm

The number of iterations for different intervals is shown in Table 1. The number of floating-point multiplications (FPM) in one iteration of *rk4* in HotSpot and in the $\mathbf{Ax} + \mathbf{Bu}$ operation is shown in Table 2. The speedup of TILTS algorithm compared to HotSpot for Pentium 4 processor is also shown in Table 2. From Table 2 we can see that for a sampling interval of 40 ms, the number of FPMs is reduced by 102400 * 4.7 = 482K times.

Usually the power estimation interval is of the order of milliseconds and our TILTS algorithm is very efficient in calculating temperature for such intervals. Therefore, with TILTS, we can significantly reduce the overhead of temperature calculations. One property of the TILTS algorithm is that the number of FPMs is fixed for one temperature calculation interval. We can thus choose different temperature calculation intervals for different purposes. To reduce the overhead of temperature calculations, we can use a longer interval, or if a more accurate temperature estimation or finer granularity is required, we can use shorter intervals.

3. A LIGHTWEIGHT RUNTIME TEMPERATURE MONITORING TOOL

Based on TILTS, we propose a temperature estimation architecture that uses the CPU internal performance counters as a proxy for CPU activities. We will call our lightweight runtime temperature monitoring tool *Temptor* (*Temp*erature Moni*tor*).

3.1 Using Temperature Sensors

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, thus are costly to implement and may even exacerbate the thermal problem by dissipating too much power. Second, since a chip usually contains only a few sensors, their placement on the chip becomes an important issue. If the sensor is not placed in one of the hot spots of the chip, then the CPU chip could become too hot without triggering CPU throttling. Third, unless the sensor is implemented in CMOS, its response time to a temperature change can be too slow which may result in late triggering of the CPU throttling in cases of thermal emergency. Digital monitoring ICs can not process diode measurements faster than 8 times/sec (conversion time is 125 ms and higher). This means that the measured temperature is lagging the real temperature of the die. Therefore, an accurate overall profile of the thermal distribution of the chip at runtime can be difficult to obtain through thermal sensors.

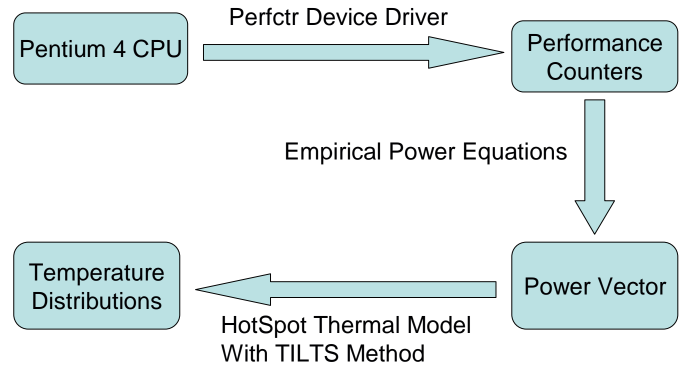


Figure 1: Framework of *Temptor* tool for Pentium 4 CPU.

Our temperature estimation method in *Temptor* does not have such limitations. It utilizes the already existing performance counters inside the CPU chip, so there is no additional cost. Our *Temptor* tool can provide detailed temperature distribution information, which is almost impossible with a small number of thermal sensors. We can also adjust the temperature calculation interval to get finer or coarser granularity. Moreover, in some processors (such as several AMD processors), there are no internal thermal sensors, but internal performance counters are provided. In such cases, our method is the only way to obtain spatial temperature information.

3.2 The Framework of Temptor

The framework of *Temptor* is shown in Figure 1. The *perfctr* device driver [24] provides physical access to the hardware counters by user-level programs. The user-level program configures and reads out the values of the hardware counters inside the CPU. These hardware counter values are then fed into empirical power equations to calculate the power consumptions of all functional units of the Pentium 4 CPU. Power equations similar to those presented in [18, 19] are used. Finally, the power consumption values are fed into the HotSpot thermal model and the temperatures of all functional units are calculated with our TILTS algorithm.

Temptor can be utilized by the operating system to provide detailed temperature readings for various DTM techniques. If a critical temperature threshold is violated, some predefined actions could be taken to bring down the temperature of the chip.

All parts of the *Temptor* framework will be described in detail in what follows.

3.3 Pentium 4 Architecture

The Pentium 4 processor has more than 42 million transistors and has more performance monitoring hardware compared to previous Pentium family processors. The Pentium 4 includes 45 configurable events and 18 physical performance counters [4]. The performance counters are used to count specific micro-architectural events relevant to performance measurements. Each hardware counter is associated with one counter configuration control register (CCCR), which determines the specific counting scheme. The event selection control registers (ESCR) determine which event is to be counted. Performance counters can be configured to gather specific micro-architectural events such as cache hits, branch prediction misses, micro-operations retired, etc. The performance events data can be used to understand how applications, the operating system, and the processor are performing, and therefore provide a good estimate of processor activity. Our *Temptor* tool uses these performance events

Table 2: **Comparison of the number of floating-point multiplications (FPM) in one iteration of *rk4* in HotSpot to that in the $\mathbf{Ax}+\mathbf{Bu}$ operation for a 40 ms interval.**

processor	N	M	# of FPM in *rk4*	# of FPM in $\mathbf{Ax}+\mathbf{Bu}$	ratio	# of FPM in HotSpot in 40 ms interval	speedup
Pentium 4 - Prescott	76	22	40918	8692	4.71	4190003200	482304

as a proxy for CPU activity, from which power consumption and temperature are derived.

3.4 The Perfctr Device Driver

Instead of using the Abyss device driver [28] as in [22], we decided to use the *perfctr* device driver developed by Pettersson on which to build our *Temptor* tool. The *perfctr* device driver [24] supports many processor models as well as many Linux kernel versions. Various performance analysis tools are based on this device driver, e.g., the PAPI (Performance Application Programming Interface) tool suite [12].

The *perfctr* library provides user-level programs convenient access to the physical performance counters. In this library, two modes of counters are provided: per-process performance-monitoring counters and global-mode (system-wide) performance-monitoring counters. The per-process counters only count the events for a particular process, which is very convenient for application profiling, performance optimization, etc. The global-mode counters count all events of the operating system, disregarding which processes they are generated from.

The temperature calculation in *Temptor* is based on the power consumption of each functional unit, to which all processes' activities contribute, so we choose to use global-mode (system-wide) performance-monitoring counters.

3.5 Performance Counters Based Power Estimation

Performance counters can be configured to gather specific micro-architectural events such as cache hits and branch prediction misses, and provide therefore a good estimate of processor activity. Isci and Martonosi have shown that accurate runtime modeling of power is possible using the performance counters [18]. Our proposed technique attempts to provide a detailed runtime micro-architecture level profile of the chip's temperature distribution.

Isci et al. have developed empirical power estimation equations for the Pentium 4 CPU. We use the same equations except for adjusting some parameters in the equations to reflect the power consumption of the Pentium 4 CPU in our experiments. They have used a 1.4GHz Pentium 4 CPU, while we use a 3GHz Pentium 4 CPU.

The power consumption of each component is estimated based on its access rate. The access rate of each functional unit is usually derived from a combination of several micro-architectural performance events. The empirical power estimation equation is as follows:

$$Power(C_i) = AccessRate(C_i) \times ArchitecturalScaling(C_i) \times MaxPower(C_i) + NonGatedClockPower(C_i) \quad (15)$$

Since not all performance metrics can be measured simultaneously using the 18 performance counters inside the Pentium 4 CPU, four phases of counter settings 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. That is, in each phase, the hardware counters are reconfigured to count different events.

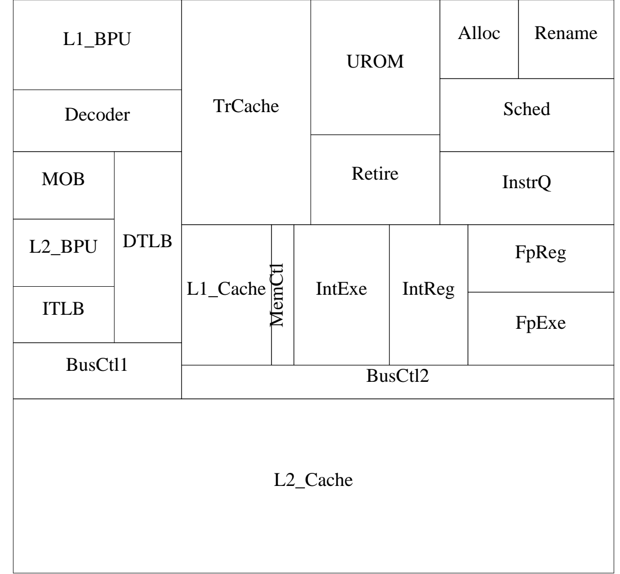


Figure 2: Floorplan for Pentium 4 Prescott die

The time of one phase is set to 10 ms in our experiments, so the power and temperature calculation interval is 40ms. The overhead of temperature calculations is very small for such a temperature calculation interval, as can be seen from Table 2. The interval of one phase can be reduced to obtain finer granularity of the temperature calculations. An interval length of 100 μs or 1 ms is possible, and not too difficult to implement.

3.6 From Power to Temperature

The Pentium 4 CPU used in our experiments is a 90nm Prescott core. Its floorplan is shown in Figure 2. The floorplan is drawn based on the annotated die photo from [2]. 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 queue, scheduler, retirement unit, floating-point execution unit, floating-point register file, integer execution unit, integer register file, and the memory control unit. The bus control unit is divided into two units in order to construct the HotSpot thermal model.

The equivalent thermal RC circuit is constructed using the HotSpot thermal model based on the floorplan in Figure 2. The HotSpot model parameters used in our experiments are shown in Table 3. For a certain temperature calculation interval, we can pre-calculate the matrices $\mathbf{A}$ and $\mathbf{B}$ used in TILTS. After that, we use Equation (13) to calculate the temperatures of the functional units from the estimated power numbers of the functional units. Therefore, we can obtain the temperature distribution of the whole CPU chip for every temperature calculation interval.

4. EXPERIMENTAL RESULTS

Table 3: HotSpot model configuration.

HotSpot parameter	value	description (unit)
t_chip	0.00074	chip thickness (m)
c_convec	140.4	convection capacitance (J/K)
r_convec	0.1	convection resistance (K/W)
s_sink	0.076	heat sink side (m)
t_sink	0.051	heat sink thickness (m)
s_spreader	0.035	heat spreader side (m)
t_spreader	0.0015	heat spreader thickness (m)
t_interface	0.000127	interface material thickness (m)
ambient	40	ambient temperature ($^{\circ}C$)

4.1 Experimental Environment

The computer we used in the experiments is a 3GHz Pentium 4 processor, which is a 90nm Prescott core. The power dissipation is 89 Watts [5, 6].

The Linux kernel 2.6.15 is installed on our Pentium 4 machine. In order to read the performance counters, the PAPI tool suite 3.2.1 [8] has been installed on the machine, which includes the *perfctr* device driver patch 2.6.x as well as the *perfctr* library.

The time step in TILTS must be constant. Variations in the length of the time step may affect the accuracy of the temperature calculations, but this effect is small and has a negligible impact on the results. We therefore do not consider this effect in our experiments.

We use 10 ms as the time for one computation phase, and every 4 phases, i.e., every 40 ms, we invoke the Ax+Bu operation to calculate new temperatures. The reason we use 10 ms as the time for one phase is that the highest resolution of timers in Linux user level space is 10 ms. For such a large interval, the overhead of our program is very small.

4.2 The Overhead of Temptor

The system overhead of *Temptor* consists of: performance , power, and thermal overhead. Current approaches are heavyweight: in [22], the performance overhead of their implementation reaches 50% for some SPEC2000 benchmarks, and the thermal overhead is as high as $14^{\circ}C$ for the *IntReg* unit.

Temptor is constantly running in the background to estimate the temperature of all functional units. Reading the 18 performance counters every 10 ms takes less than 1 μs, which is negligible. Due to the use of the efficient TILTS algorithm, the temperature calculations use only $\mathbf{Ax}+\mathbf{Bu}$ operations every 40 ms, therefore, the overhead of *Temptor* is very small.

4.3 Preliminary Results

We use SPEC2000 benchmarks [3] in our experiments. The benchmarks are compiled using the "intel-linux" configuration and the SPEC "base" tuning option. The SPEC2000 benchmarks are listed in Table 4. The reference data input and the command lines used in our experiments are shown in Table 4. In each experiment, we first let *Temptor* run for some time to reach the steady state, and then run the SPEC2000 benchmarks to completion.

The maximum case temperature for the Pentium 4 CPU is $65^{\circ}C$ [6]. A temperature of $41^{\circ}C$ when the operating system is idle and a temperature of $75^{\circ}C$ when the operating system is at full load have been observed [7]. Usually there is a constant temperature difference between the CPU temperature and the temperature sensor reading of the motherboard, which can be as large as $20^{\circ}C$. Taking into account the temperature difference, the estimated temperatures in our experiments are in the range of 68-92$^{\circ}C$, which is in the reasonable temperature range.

Through several experiments, we obtained some preliminary numerical results. We plot the temperature estimates for the three hottest units: *IntReg*, *FPReg*, *Rename* (see Figures 3-7). We also plot the average temperature of the whole chip for all the benchmarks. Since the results for the remaining 9 benchmarks are similar, We only show the results for 5 out of the 14 benchmarks.

We have observed the following:

1. Thermal throttling is necessary for contemporary high performance processors. The maximum temperature in the chip when the operating system is idle is about $66^{\circ}C$, and the hot spot is the *Rename* unit. The maximum temperature when the *gcc* benchmark is running can be as high as $114^{\circ}C$(see Figure 4). The difference in the maximum temperature when the operating system is idle and some applications are runing is nearly $50^{\circ}C$.

2. The thermal behaviors of different applications are very different. Let us look at the maximum temperatures in the chip for different applications. For the *gzip* benchmark, the maximum temperature reaches $100^{\circ}C$, and the average maximum temperature averaged over the entire execution is about $92^{\circ}C$. For the *gcc* benchmark, the maximum temperature in the chip reaches $114^{\circ}C$, and the average maximum temperature over its whole execution is about $88^{\circ}C$. While for the *art* benchmark, the maximum temperature only reaches $69^{\circ}C$, the average maximum temperature over its entire execution is $68^{\circ}C$. The difference in the maximum temperature observed is $45^{\circ}C$ (*gcc* vs. *art*), and the difference between the average maximum temperature has been as large as $24^{\circ}C$ (*gzip* vs. *art*).

3. There are some common thermal behaviors among the integer benchmarks and the floating-point benchmarks. Let us look at the hot spot location in the chip. Usually the *IntReg* unit is the hot spot for the integer benchmarks, and the *Rename* unit is the hot spot for the floating-point benchmarks in our experiments. The *mcf* benchmark is an exception. It is an integer benchmark, but its hot spot is the *Rename* unit, not the *IntReg* unit. The temperature of the *Rename* unit in the *mcf* benchmark is $8^{\circ}C$ higher than the *IntReg* unit, as can be seen from Figure 5.

4. The maximum temperature for the integer benchmarks is usually greater than the floating-point benchmarks. The access rate of the *IntReg* unit in the integer benchmarks is larger than that in the floating-point benchmarks, thus more likely to become hot in the integer benchmarks. This behavior has also been observed in [16, 26].

5. The hot spot location of one particular program changes during its execution. For example, in the *gcc* benchmark, although the *IntReg* unit is the hot spot during most of the execution time, the *Rename* unit is the hot spot during a period of time. This can be seen from Figure 4. This behavior has also been observed in [22].

6. The execution of most of the floating-point benchmarks exhibits repetitive characteristics. This is probably because floating-point benchmarks quite often repeat similar computations. This characteristic can be clearly seen from Figure 7. The period length is about 11 seconds for the *apsi* benchmark in Figure 7. Most of the integer benchmarks do not exhibit a similar repetitive characteristic although the *mcf* benchmark does exhibit some repetitive behavior as seen in Figure 5.

7. As a result of the repetitive characteristic of the floating-point benchmarks, some functional units (e.g., *IntReg*, *FPReg*) expereince a large temperature swing (more than 20 $^{\circ}C$) in a short time (less than 1 second) during the execution of some floating-point benchmarks. In contrast, for most integer benchmarks, the temperature changes of functional units in a short time are small (less than

Table 4: The integer and floating-point SPEC2000 benchmarks used in our experiments

benchmark	running time	description	command line
gzip	60.8 s	Compression	gzip input.source 60
vpr	239.2 s	FPGA Circuit Placement and Routing	vpr net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2
gcc	78.0 s	C Programming Language Compiler	cc1 200.i -o 200.s
mcf	422.4 s	Combinatorial Optimization	mcf inp.in
crafty	270.0 s	Chess Game Playing	crafty < crafty.in
bzip2	111.2 s	Compression	bzip2 input.source 58
twolf	747.2 s	Place and Route Simulator	twolf ref
swim	575.2 s	Shallow Water Modeling	swim < swim.in
mgrid	549.6 s	Multi-grid Solver: 3D Potential Field	mgrid < mgrid.in
applu	644.0 s	Parabolic/Elliptic Partial Differential Equations	applu < applu.in
art	664.0 s	Image Recognition/Neural Networks	art -scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10
equake	199.2 s	Seismic Wave Propagation Simulation	equake < inp.in
ammp	895.6 s	Computational Chemistry	ammp < ammp.in
apsi	755.6 s	Meteorology: Pollutant Distribution	apsi

Table 5: Statistics of the 14 benchmarks

benchmark	avg temp ($^\circ C$) of hot spot unit in most of the run time	max temp ($^\circ C$) ever reach and the unit
gzip	92.4(*IntReg*)	99.6(*IntReg*)
vpr	74.3(*IntReg*)	91.1(*IntReg*)
gcc	83.8(*IntReg*)	114.2(*IntReg*)
mcf	68.1(*Rename*)	77.1(*IntReg*)
crafty	83.0(*IntReg*)	87.7(*IntReg*)
bzip2	86.6(*IntReg*)	95.7(*IntReg*)
twolf	76.7(*IntReg*)	88.2(*IntReg*)
swim	70.3(*Rename*)	72.8(*IntReg*)
mgrid	73.4(*Rename*)	77.4(*IntReg*)
applu	72.9(*Rename*)	84.5(*IntReg*)
art	68.3(*Rename*)	69.3(*Rename*)
equake	71.8(*Rename*)	82.2(*IntReg*)
ammp	70.0(*Rename*)	73.0(*FPReg*)
apsi	72.6(*Rename*)	84.2(*IntReg*)

10 $^\circ C$).

Some statistics of the 14 benchmarks are shown in Table 5. Notice the considerable difference between the peak temperature and the average temperature of the hot spot block. The average deviation of the instantaneous (estimated) temperature from the average temperature for all functional units is shown in Table 6. From this table we can see that the temperature variation of the *IntReg* unit is larger than other functional units.

4.4 A Thermal Stress Program

We wrote a C program *IntRegStress* to stress the *IntReg* unit. The following is the C source code of the *IntRegStress* program:

```
void main(int argc, char ** argv) {
int data[1024]; int i,j; int loops=100000000;
 for(j=0; j<loops; j++) {
  for(i=0; i<128; i++) {
   for(data[i]=0; data[i]<100; data[i]++) {
    data[i+1] ++;
   }  } }  }
```

The temperature of the *IntReg* unit when the *IntRegStress* program is running can reach $115^\circ C$. Thermal throttling is important

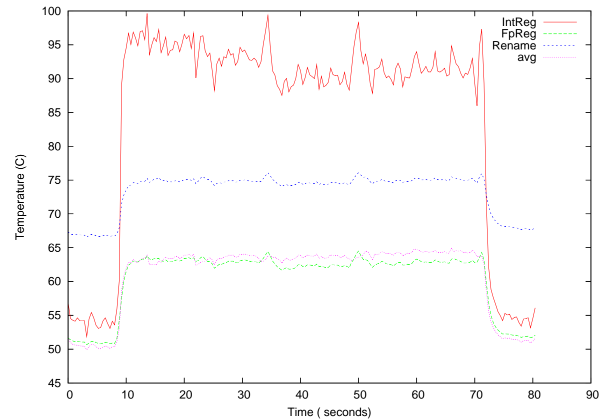


Figure 3: Temperatures for *gzip* benchmark

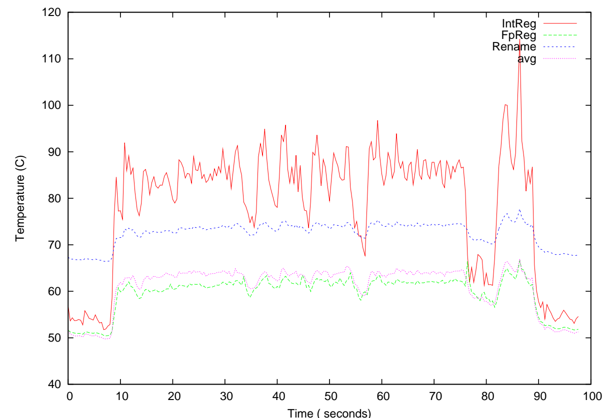


Figure 4: Temperatures for *gcc* benchmark

Table 6: Average deviation of the estimated temperature from the average temperature ($^\circ C$) for all functional units and for all 14 SPEC2000 benchmarks

bench	*gzip*	*vpr*	*gcc*	*mcf*	*crafty*	*bzip2*	*twolf*	*swim*	*mgrid*	*applu*	*art*	*equake*	*ammp*	*apsi*	avg
BusCtl1	0.6	0.4	0.6	1.1	0.6	0.6	0.4	0.5	0.4	0.5	0.3	0.9	0.5	0.9	0.59
BusCtl2	0.4	0.4	0.9	0.9	0.7	0.9	0.4	0.5	0.5	0.7	0.3	0.7	0.5	1.1	0.64
L2_Cache	1.3	0.5	1.0	0.7	0.5	0.4	0.4	0.3	0.4	0.4	0.2	0.4	0.3	0.5	0.52
L2_BPU	1.7	0.9	4.9	1.3	0.9	3.2	0.9	1.3	1.1	2.0	1.3	1.2	1.3	1.9	1.71
Decoder	0.6	0.8	3.0	0.6	1.0	1.2	0.3	0.7	0.5	0.6	0.9	0.7	0.6	1.4	0.92
L1_BPU	0.9	0.6	2.5	0.6	0.7	1.6	0.5	0.6	0.6	0.9	0.6	0.6	0.6	0.9	0.87
ITLB	0.8	0.5	1.7	0.6	0.6	1.3	0.5	0.5	0.5	0.6	0.4	0.7	0.5	0.8	0.71
MOB	0.7	0.6	2.0	0.5	0.7	1.5	0.4	0.6	0.5	0.7	0.5	0.6	0.9	1.0	0.80
TrCache	0.5	0.4	1.6	0.5	0.8	1.3	0.5	0.4	0.5	0.7	0.4	0.4	0.5	1.7	0.73
DTLB	0.6	0.5	1.8	0.6	0.8	1.6	0.5	0.5	0.5	0.7	0.5	0.8	0.9	2.1	0.89
L1_Cache	0.7	0.6	2.5	1.0	1.1	2.1	0.6	0.8	0.8	1.3	0.6	1.1	0.9	3.9	1.29
IntExe	1.6	0.7	4.4	1.3	1.3	3.1	0.7	2.1	1.6	2.8	1.2	1.4	1.5	4.5	2.01
MemCtl	0.8	0.5	2.7	0.9	1.0	2.1	0.5	1.0	0.8	1.4	0.6	0.8	0.8	3.1	1.21
IntReg	2.1	0.9	5.5	1.7	1.5	3.9	0.8	2.7	2.0	3.8	1.6	1.8	1.9	5.8	2.57
FpExe	0.3	0.5	0.9	0.5	0.7	0.9	0.4	1.3	1.0	2.4	0.6	0.9	2.2	1.5	1.01
FpReg	0.4	0.5	1.1	0.4	0.8	1.0	0.5	1.5	1.1	2.6	0.6	1.0	2.6	1.6	1.12
UROM	0.7	0.2	0.8	0.2	0.7	0.6	0.5	0.3	0.4	0.4	0.2	0.5	0.3	0.7	0.46
Alloc	0.3	0.3	0.9	0.3	0.7	0.7	0.4	0.3	0.4	0.5	0.2	0.2	0.4	0.9	0.46
Rename	0.3	0.3	0.9	0.3	0.7	0.7	0.4	0.3	0.4	0.5	0.2	0.2	0.4	0.9	0.46
Retire	0.6	0.4	1.8	0.5	0.8	1.3	0.5	0.6	0.5	0.8	0.4	0.4	0.5	1.8	0.78
InstrQ	0.5	0.4	1.6	0.5	0.8	1.3	0.5	0.5	0.5	0.8	0.4	0.4	0.9	1.7	0.77
Sched	0.3	0.3	0.9	0.2	0.7	0.7	0.4	0.3	0.4	0.4	0.2	0.2	0.4	0.8	0.44

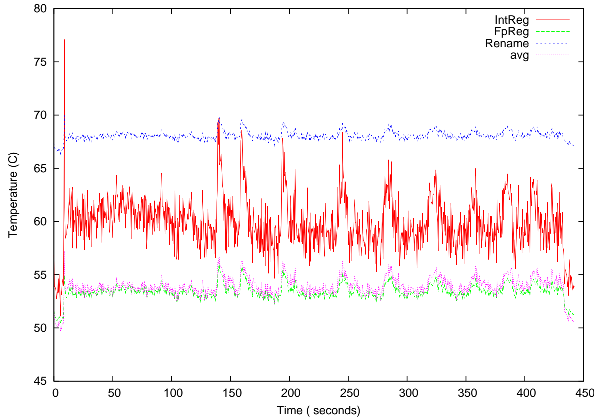


Figure 5: Temperatures for *mcf* benchmark

to deal effectively with software that severely stresses individual units of the architecture.

4.5 Future Research

In the future, we plan to read out the temperature sensor values and calibrate the HotSpot thermal model. Such a calibration will improve the accuracy of the *Temptor* tool.

Understanding application-specific thermal behavior can guide researchers in designing new thermal-aware techniques, floorplan layouts, and in determining the placement of thermal sensors on the chip. In the future, we will explore the thermal behaviors of multimedia applications, such as MediaBench [21]. We can use it on SMP computers, and in distributed environments. *Temptor* will be made available to other researchers in the near future.

5. CONCLUSIONS

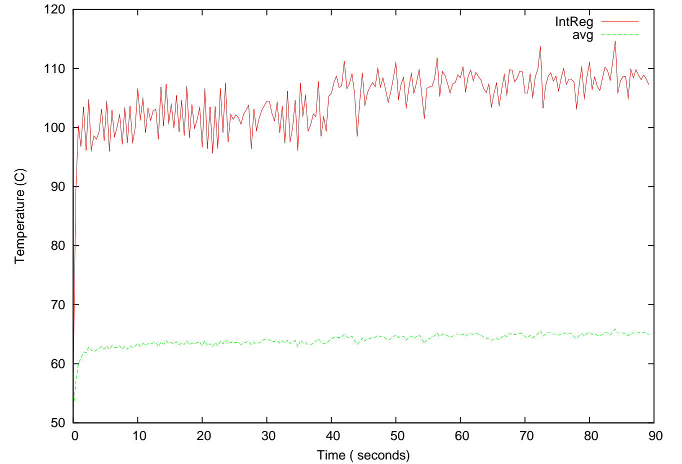


Figure 8: Temperatures for *IntRegStress* program

We have presented in this paper a new transient thermal simulation algorithm, TILTS, which is much faster than conventional simulation algorithms. Based on the TILTS algorithm, we have developed a lightweight runtime temperature monitoring tool called *Temptor*. Using internal performance counters, *Temptor* is able to estimate runtime temperature distributions in CPU chips. Even in the absence of on-chip temperature sensors, *Temptor* can obtain a detailed spatial temperature distribution through software modeling.

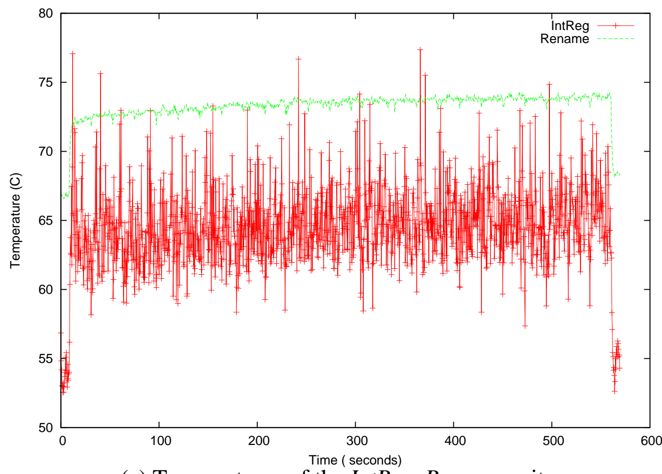
Temptor uses hardware performance counters to measure processor activities and provides detailed temperature information at the architecture level. Most importantly, the bottleneck of inefficient temperature calculations in [22] is resolved in *Temptor*. The system overhead of *Temptor* is negligible. *Temptor* can be used for high-performance applications that require low-overhead temperature sensing for runtime DTM techniques. *Temptor* provides us a real-time experimental platform to study various temperature aware techniques. Our experimental results show that *Temptor* enables us to study many thermal issues in high performance microprocessors, which is complementary to purely software simulation based approaches.

6. ACKNOWLEDGEMENTS

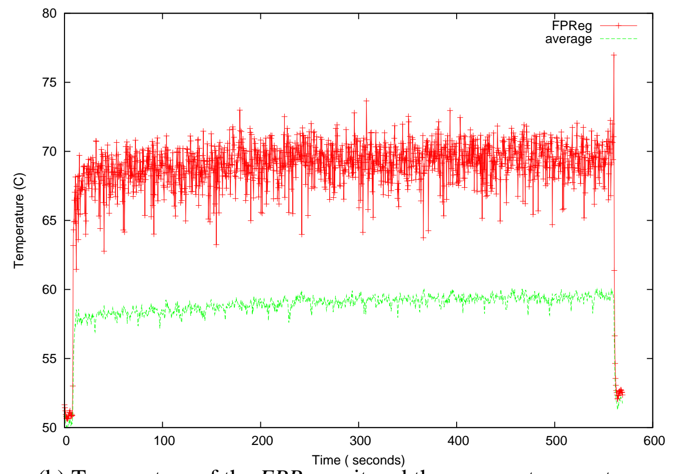
The authors wish to thank Mikael Pettersson at Uppsala University for the *perfctr* device driver to read performance counters, and Canturk Isci at Princeton University and Kyeong-Jae Lee at the University of Virginia for providing their source codes for estimating power and temperature.

7. REFERENCES

[1] http://lava.cs.virginia.edu/hotspot.
[2] http://www.chip-architect.com/news/.
[3] http://www.spec.org.
[4] IA-32 intel architecture software developer's manuals.
[5] Intel pentium 4 processor on 90 nm process datasheet.
[6] Intel pentium 4 processor thermal management.
[7] Pentium 4 cpu temperature, http://forums.vnunet.com/thread.jsp?forum=3&thread=11363.
[8] Performance application programming interface, http://icl.cs.utk.edu/papi/index.html.
[9] Intel tries to keep its cool. *PC World*, april 2004.
[10] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the Seventh International Symposium on HighPerformance Computer Architecture (HPCA-7)*, pages 171–182, 2001.
[11] 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 (ISCA)*, pages 83–94, 2000.
[12] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.
[13] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, 1997.
[14] P. Chaparro, J. González, and A. González. Thermal-aware clustered microarchitectures. In *International Conference on Computer Design (ICCD)*, pages 48–53, 2004.
[15] M. Gomaa, M. D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 260–270, New York, NY, USA, 2004. ACM Press.
[16] Y. Han, I. Koren, and C. A. Moritz. Temperature aware floorplanning. In *Second Workshop on Temperature-Aware Computer Systems(TACS-2)*, June 2005.
[17] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusam. Compact thermal modeling for temperature-aware design. In *Proceedings of the 41st Annual Conference on Design Automation (DAC)*, pages 878–883, 2004.
[18] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *36th ACM/IEEE International Symposium on Microarchitecture (MICRO-36)*, pages 93–104, December 2003.
[19] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. technical report. December 2003.
[20] C. Isci and M. Martonosi. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. In *12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, pages 121–132, Febrary 2006.
[21] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.
[22] K.-J. Lee and K. Skadron. Using performance counters for runtime temperature sensing in high-performance processors. In *IPDPS*, April 2005.
[23] A. V. Oppenheim, A. S. Willsky, and N. S. Hamid. Signals and systems, 1996.
[24] M. Pettersson. Perfctr device driver, http://user.it.uu.se/~mikpe/linux/perfctr/.
[25] K. Skadron, T. Abdelzaher, and M. R. Stan. Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. Technical Report CS-2001-27, 2001.
[26] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 2–13, 2003.
[27] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, 2004.
[28] B. Sprunt. Brink and abyss pentium 4 performance counter tools for linux, July 2004.

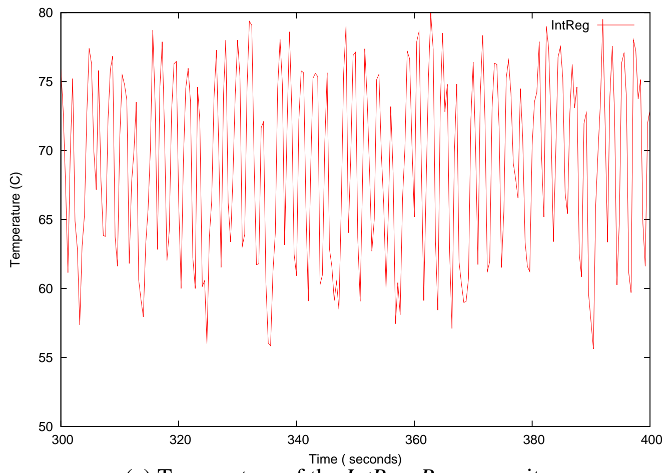


(a) Temperatures of the *IntReg*, *Rename* units

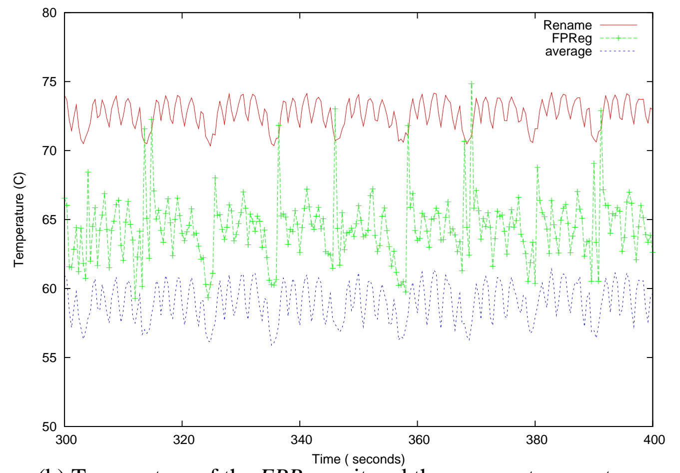


(b) Temperature of the *FPReg* unit and the average temperature

Figure 6: Temperatures for *mgrid* benchmark



(a) Temperature of the *IntReg*, *Rename* units



(b) Temperature of the *FPReg* unit and the average temperature

Figure 7: Temperatures for *apsi* benchmark