# Dynamically Managing Processor Temperature and Power

Erven Rohou and Michael D. Smith
Harvard University
33 Oxford Street
Cambridge, MA, 02138, USA

## Abstract

Hardware designers are facing the following dilemma: they must ensure that the processor temperature will never exceed a safe maximum, but they also know that this maximum is reached only under unrealistic benchmarks. In other words, the processor could be more efficient for an average workload. Maintaining a safe temperature bound is made difficult because it depends on system statistics as well as external parameters such as the room temperature.

We present an adaptive approach that uses feedback to keep the processor temperature in a safe range. The temperature is regularly sampled. When it reaches a dangerous level, the applications responsible are slowed down. Our technique is implemented in the operating system so that it can both access hardware statistics and control the interleaving of processes. This allows us to affect only CPU-intensive (or "hot") processes and not reduce the responsiveness of interactive processes.

We are able to keep the temperature under a predefined threshold by constraining the maximum allowed CPU activity. This approach is superior to throttling: it does not affect slow processes and it has a better resolution when choosing a slowdown ratio. Moreover, near the temperature maximum, our results show that we can significantly reduce the temperature with little cost in performance.

## 1 Introduction

Building systems that consume less power has been an important research topic in the computer community over recent years. This need originated in the embedded systems world and has appeared in the design of commodity processors as a way to extend battery life in laptops and to address the call for environmentally-friendly desktops. The continuous trend toward higher levels of VLSI integration and faster clock frequencies means that control over power dissipation will become an even more important issue in the design of future microprocessors.

Maintaining the temperature of a processor below a safe operating limit is a problem related to power consumption, since high-power devices produce more heat. However, there are differences between the desire to remain below a power bound and the need to remain below a temperature bound. If the processor in your laptop exceeds its power bound, your battery dies sooner. If the processor in your laptop exceeds its temperature bound, your processor dies.

To ensure that the processor temperature will never exceed a safe maximum, designers run benchmarks to characterize the worst-case scenarios. These scenarios often comprise unrealistic workloads that a typical user would never encounter. Since the design of modern systems do not have any way to guarantee that these unrealistic workloads will never occur, a conservative approach is required. The result is that these artificial benchmarks drive the design of the processor, resulting in less efficient hardware, though more efficient designs would be safe for real applications.

At the same time, people like to overclock their processors. To overcome the glitches caused by running the chip faster than it was rated, they also increase the supply voltage. The combination of higher voltage and frequency result in a much higher power dissipation, and therefore processor temperature.

In this paper we present an adaptive technique that uses feedback to keep the processor temperature below a safe bound. As we will discuss, the processor temperature depends on several parameters, including external parameters such as the temperature of the room and the speed of the fan. In our approach, the processor temperature is regularly sampled. When it reaches a dangerous level, our system targets the responsible applications and slows them down. Though designed to control temperature, we can also use our technique to control power consumption.

In particular, we implemented our approach as a software technique that allows the operating system to control CPU activity on a per-application basis. By slowing down CPU-bound applications, we can control both power and temperature in a very accurate way: placing

*Published at the 2nd Workshop on Feedback-Directed Optimization (FDO-2) 1999, in Conjunction with MICRO-32.*

the control in the operating system allows us to slow down only "hot" processes without affecting infrequent and/or interactive processes. Temperature feedback from the processor helps determine the best safe CPU activity level. As our results demonstrate, we are able to maintain any temperature constraint. By letting the software ensure that the temperature constraints are met, we remove the need for the conservative approach to hardware design and potentially enable the manufacturing of faster processors.

Section 2 reviews related work. We detail our technique in Section 3, give results in Section 4, and conclude in Section 5.

## 2 Related Work

Sanchez et al. [15] describe a temperature sensor implemented in the PowerPC microprocessor. The absolute error is within $\pm 12°$C and the sensor can only be used as a safety mechanism to avoid catastrophic events. The Pentium II and III also feature thermal diodes in their core [9]. With the appropriate hardware on the motherboard, an application can read the temperature of the die. Even though several utilities exist to monitor the temperature of the processor, we are not aware of any previous work related to controlling this temperature. Yet this is considered a critical issue by those in industry who need to guarantee that their product will meet the temperature requirements [10]. Since power consumption and temperature are not independent, both are monotonically increasing functions of CPU activity, this section reviews the literature on low power techniques. A technique known to reduce power consumption is expected to also reduce the temperature.

Every component in a computer system (processor, memory, devices) draws current from the main power supply. Devices usually reduce their power consumption by entering *standby* or *sleep* modes when they have been unused for a period of time. ACPI (Advanced Configuration and Power Management Interface) [11] is a specification that is intended to standardize the different power levels of a computer system and the different ways an operating system should interact with the hardware. ACPI defines four levels (D0 to D3) for devices. The larger the level number, the lower the power consumption, but also the longer the response time. Similar levels are defined for the processor (C0 to C3), the global system (G0 to G3), and sleep modes (S1 to S5).

We can reduce the power consumption of a processor by reducing the supply voltage, the capacitance, or the clock frequency. Liu and Svensson [14] trade supply voltage against speed. Dynamically changing the capacitance requires hardware support to shut down parts of processors [1]. Reducing the clock frequency is often referred to as *throttling*. This is done by modern PCs when the system has been idle for a while. Each of these techniques affect all running processes, not just those consuming the most power.

Memory dissipation can be reduced by decreasing the number of memory accesses. Usual algorithms to improve data locality apply: Zervas, Masselos and Goutis [20] use loop transformations to modify the data access patterns and show power savings in the range of 30–50 % for multimedia applications. Gebotys [8] studies the impact of memory and register allocation on memory dissipation and finds an improvement factor between 2.8 and 4.9. Lee et al. [12, 13] use simulated annealing to partition data between the two memory banks of a Fujitsu DSP so that the two operands of an ALU operation can be loaded in parallel. They report an energy reduction in the range of 23 % to 47 %.

Several authors address the switching probability. Lee et al. [12] and Su et al. [17] modify a list scheduler to take into account the power consumed by the execution of any pair of instructions (*cold scheduling*). Su et al. [17] also propose to use the Gray code to encode addresses on data and instruction buses. They achieve a 37 % reduction in the number of bit-switches per executed instruction. Tiwari, Malik and Wolfe [18] report a very small improvement with the same algorithm on a different architecture. Toburen, Conte and Reilly [19] schedule operations for an 8-issue VLIW processor. Energy dissipation is computed at each cycle. When a predefined threshold is reached, the scheduler quits scheduling the current cycle and begins scheduling for the next cycle with the instruction that caused the violation. Results show significant energy savings with little impact on performance. Chang and Pedram [3] present a technique to reduce the external switching activity of a register file by register allocation. They use probabilistic density functions as input and they use graph theory to solve the problem.

Though compile-time techniques can produce power-conscious applications, they cannot guarantee that an application will not cause the processor to exceed its temperature bound. Furthermore a system that relies on these techniques to maintain its temperature below a safe bound can run only temperature-aware applications.

## 3 Dynamic Control of Temperature

Many existing techniques aimed at reducing power consumption consist of static optimizations. We propose instead a dynamic approach to controlling the temperature of the processor and its power dissipation. We first present the technical context on which our work is based. Then we discuss the principle of our approach and its implementation.

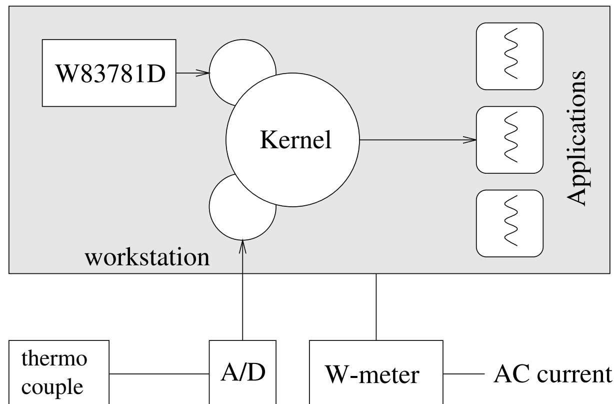



Figure 1: Experimental setup.

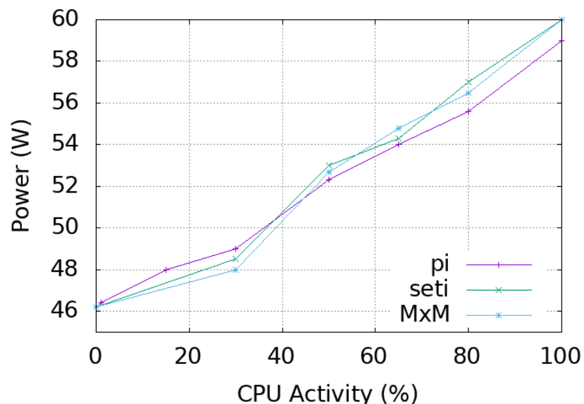



Figure 2: Power vs CPU activity. The benchmarks are: computation of digits of $\pi$ (pi), matrix multiplication (MxM), and a signal processing application (seti).

### 3.1 Preliminary Study

Recent Intel processors provide the `HLT` instruction [4] to stop instruction execution and place the processor in a HALT state until an interrupt occurs. This state corresponds to the C1 state of the ACPI specification: all of the processor except the interrupt logic is shut down, resulting in a significant energy saving.

The Linux kernel executes this instruction when no process is runnable, thus reducing the power consumption when no computation can be performed. Our Pentium II workstation dissipates 46 W when idle and 60 W under heavy workload. Similarly a laptop ranges from 17 W to 40 W. When the `HLT` instruction is disabled (or when running an operating system that does not exploit this feature, such as MSDOS or Windows98), both computers consume the maximum value. Using `HLT` respectively yields 23 % and 57 % energy savings on idle machines.

Power consumption and temperature are two related issues. However, they have still have their own characteristics, which we discuss in the next paragraphs.

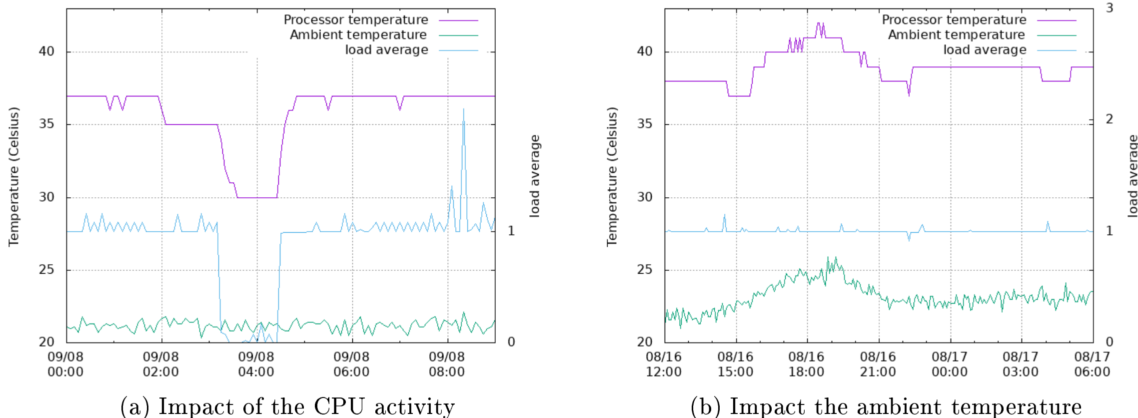
In the following, the expression “CPU activity” denotes the ratio between the time spent running processes and the real time elapsed. This number is reported under UNIX by the popular commands `ps` and `top`. This number is in the range 0–100 %. The “load average” is the average number of runnable processes during the last minute. It is commonly displayed by the tool `xload`.

**Power consumption** Our experimental setup consists of a Pentium II workstation running Linux 2.2.10 (see Figure 1). Power consumption is obtained with a wattmeter inserted between the power supply of the workstation and the AC current plug. The readings measure the consumption of the whole desktop box, i.e. processor, motherboad, disks. We assume that the consumption of all parts but the processor is constant and that variations reflect only changes in the processor consumption. This is realistic since our workload is CPU- or memory-bound and does not involve any disk or floppy access.

Due to the use of the `HLT` instruction, the power consumption of the processor is directly related to the CPU activity. We tested several kinds of applications, ranging from intensive computation kernels with no memory accesses to manipulation of large data sets, involving a lot of memory accesses, without noticing any significant difference in the power requirement. Instruction scheduling also made little difference. All the variations were within a $\pm 2$ % range. All that mattered was whether the CPU was active or not. Figure 2 illustrates the variation of the power consumption versus the CPU activity for the Pentium II workstation under various workloads.

Because of this application independence, we ran our experiments with only one benchmark: the signal processing application, part of the SETI project [6].

**Temperature** The motherboard features an Winbond W83781D thermal sensor [5] that can be used to take temperature readings of the processor at any time (see Figure 1). Because the sensor is not directly touching the processor case, the time constant of our system is larger that it could be and the temperature range is narrower. However, direct measurement of the processor case temperature with an infrared thermocouple (whose time constant is 60 ms) shows that the variations do not ex-




(a) Impact of the CPU activity

(b) Impact the ambient temperature

Figure 3: Variations of the processor temperature.

ceed 30°C in 10 seconds, which still leaves ample time for software-based control. The ambient temperature is available thanks to an external thermocouple connected to the PC through an analog-digital converter.

The main characteristic of temperature is that it *must* stay under a predefined threshold. Exceeding it may damage the processor. Hardware designers must be very careful that the temperature can *never* reach an unsafe point. For this reason, processors usually do not reach the performance that the current technology would allow; some artificial benchmark can be built that exceeds the temperature limit.

As with power, the temperature of the processor depends on CPU activity. Figure 3(a) shows the variation of the temperature over time, along with the load average[1] of the machine. When the load average drops from 1 to 0 (the running program stopped) shortly after 3am, the temperature rapidly falls down by 5°C. The start of a new application one hour later is followed by a temperature increase of 7°C.

The temperature of the room where the computer is located also has an impact. If air-conditioning is not available in this room, the processor temperature cannot be accurately estimated from system statistics. Figure 3(b) illustrates the impact of the ambient temperature on the processor, while the load average remains constant. The processor temperature is initially 38°C at noon. In the late afternoon it reaches 42°C. This clearly shows that a static approach will not be able to maintain the temperature under the threshold without being overconservative. Even with profiling information, it cannot take into account all parameters. A dynamic approach, using temperature feedback, is clearly needed to maintain the temperature in a safe range without being excessively conservative.

[1] In this paper, running processes are compute-bound: a load average equaling 1 means a process running at full speed.

## 3.2 Principle of our Approach

The actual power consumption and temperature result from the interaction of applications with the hardware. However, given the organization of today's systems, independent applications usually have no way to communicate with each other, and it is difficult for code running in user space to get information from the hardware. On the other hand, the operating system has access to low level hardware statistics and controls the execution of applications. For these reasons, the operating system is the only sensible place to implement our technique.

Run-time optimization systems have been developed in recent years [7]. They postpone some optimizations until the execution of an application. The behavior of the application is monitored, and it can trigger specific optimizations that depend on specific run-time values. We simply applied these ideas to power and temperature rather than performance.

We modified the kernel of to introduce the concept of "hot" process. A process is said to be hot when it uses more than a specific CPU activity over a period of time. In this case it is responsible for a higher power consumption and for a temperature increase; it may thus be subject to a penalty. Our algorithm to maintain the processor temperature under a safe bound is illustrated in Figure 4. When the temperature is below the bound, the system works in the usual way and simply samples the current

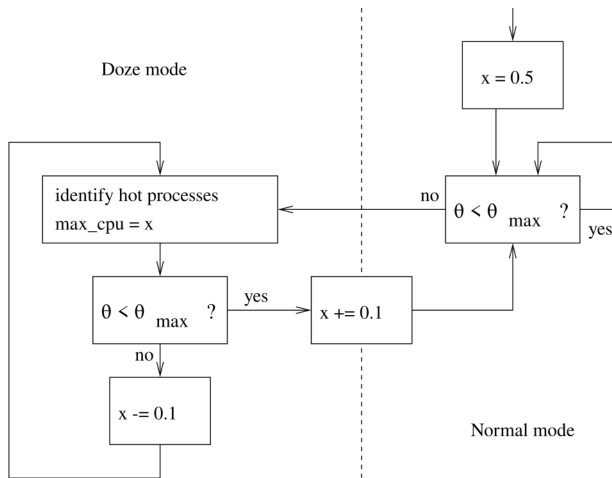



Figure 4: Algorithm used to maintain the processor temperature under a safe bound. $\theta$ denotes the current temperature, $\theta_{max}$ the bound, and $x$ the maximum allowed CPU activity for hot processes.

processor temperature (right hand side of the figure). As soon as a threshold is reached the system enters a special mode, which we call *doze mode* (left hand side of the figure): hot processes are identified and slowed down by an appropriate amount. In other words these processes are not allowed to consume more time than a fraction $x$ of the time they would normally use (initially the fraction is $x = 0.5$). The hardware is then put under observation for a period of time. If the temperature criterion is met at the end of this period, the system exits doze mode and returns to normal operation. However, if the temperature is still too high, the maximum allowed CPU activity is decreased by 0.1 and the observation period starts again.

Eventually, the processor temperature drops below the bound and control returns to normal mode. When this occurs, we increase $x$ by 0.1. This allows the system to find a fraction that successfully controls temperature without excessively reducing performance. Though simple, this algorithm is efficient and conceptually similar to algorithms found in other fields, e.g, the congestion-avoidance algorithm in TCP/IP [16].

By dynamically updating the activity of hot processes, we can precisely adjust the highest CPU level the system can safely support. The system also automatically reacts to newly created processes or dead processes by regularly updating the list of hot processes and sharing the maximum allowed CPU between them. Furthermore I/O bound and/or interactive processes are not affected and still run at full speed.

### 3.3 Implementation

Our technique is implemented in the Linux kernel. All the changes we made to the system are located in the files `/usr/src/linux/kernel/sched.c` which implements the process scheduler and `/usr/src/linux/include/linux/sched.h` which contains the definition of the `task` structure.

Every process maintains two priority numbers that control its speed in the system: the static priority (also known as the *nice value*) and the dynamic priority that specifies how many clock ticks the process can live before its time slice expires. When a process has consumed its time slice, it is given a new one according to its static priority. Refer to Beck at al. [2] for details on the Linux scheduler.

We added two entries to the `task` structure:

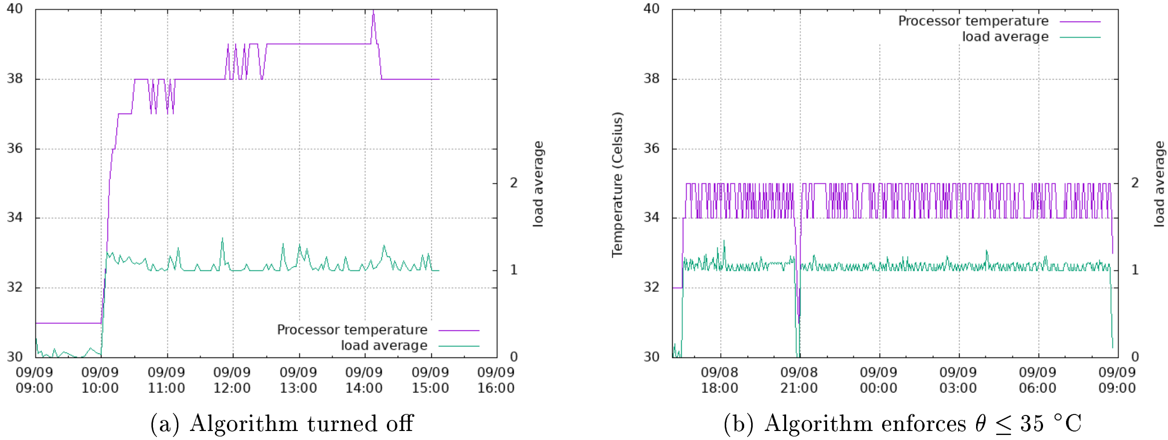
- Each process records its activity over a short period of time. This allows us to have an up-to-date list of hot processes that takes into account possible changes in the processes behaviors. For example a process that starts reading a file or writing results may become I/O-bound, and is no longer hot. Conversely, a process that was stopped on a blocking read can start intensive computations and suddenly become a hot process.
- Each hot process is assigned a timestamp before which it is not allowed to execute. The value of this timestamp is a function of the current slowdown factor and is updated whenever the process exits run-mode. The greater this number, the slower the process. We experimentally determined the delays corresponding to various slowdown factors.

Note that changing the nice value is not a solution, since a process running at a very low priority can still run the CPU continuously if it is the only runnable process.

Whenever the scheduler recomputes the dynamic priorities of the processes, the system takes a reading of the processor temperature and may enter doze mode. If doze mode is enabled, the system then checks for the end of the current observation period and accordingly increases or decreases the maximum allowed CPU activity. Finally, hot processes are identified and their execution is postponed according to the current system state.

## 4 Results

To validate our technique, we used it to maintain power consumption and processor temperature under various levels. Sections 4.1 and 4.2 detail these experiments. Section 4.3 studies the impact of our technique on the performance of applications.




(a) Algorithm turned off

(b) Algorithm enforces $\theta \leq 35$ °C

Figure 5: Impact of our algorithm on the temperature of the processor.

### 4.1 Power Consumption Control

Power consumption does not depend on any external parameter. In this simple case, no feedback is needed. To achieve a given level, the system simply performs a table lookup to determine the corresponding maximum allowed CPU activity (or slowdown ratio), shown in Figure 2. This number in turn translates into a number of clock ticks that is assigned to each hot process.

Our experiments show that any given power constraint can be met, sometimes at the expense of a significant slowdown of running applications.

### 4.2 Temperature Control

The temperature of the processor is much more difficult to predict. It depends on various parameters, such as the ambient temperature, the speed of the fan and the size of the case. Figure 5(a) illustrates the increase in processor temperature when a compute-bound process starts and no control is applied. We see an increase of $+7$°C after a few minutes, and a peak at $+9$°C after several hours. We ran the same program again while constraining the temperature to remain under 35°C. Figure 5(b) shows the result. The temperature oscillates between 34 and 35 degrees. Note that the resolution of the W83781D thermal sensor is $\pm 1$°C. The sudden decrease around 9pm can be explained as follows: our application downloads a chunk of data to process over the Internet once or twice a day. The server was down for a few minutes and our client was stopped on a blocking read.

We obtained similar results with different temperature threshold. These experiments show that implementing a temperature control in the operating system provides a safe solution.

### 4.3 Slowdown

Our technique guarantees that a given threshold cannot be exceeded. We are now interested in the resulting delays. Figure 6 shows the maximum allowed CPU activity and the current processor temperature over time. It illustrates the oscillating behavior of the system between normal (max_cpu=100 %) and doze (max_cpu $<$ 100 %) mode, and the system's attempt to reduce the temperature by iteratively reducing the available CPU by 0.1. This staircase effect is clearly visible around 9pm: the sudden stop of the program is followed by a temperature decrease and a prompt return to normal mode. When the temperature goes up again, the system enters doze mode and chooses a maximum CPU activity equal to 0.9, i.e. the last one used (0.8) plus 0.1. Since it is not enough, the system keeps decreasing the activity down to 40 %. The overall average available CPU is 73 % on Figure 6.

Figure 7 plots the daily average available CPU as a function of the maximum allowed temperature. A temperature limit is specified to the system and the variations of the maximum CPU activity max_cpu(t) are recorded (as in Figure 6). The average is then computed as

$$||\text{max_cpu}||_1 = \frac{1}{24h} \int_{t_0}^{t_0+24h} \text{max_cpu}(t)dt$$

As expected, the higher the temperature, the better the performance. It should also be noted that a variation of one degree in the maximum temperature has larger

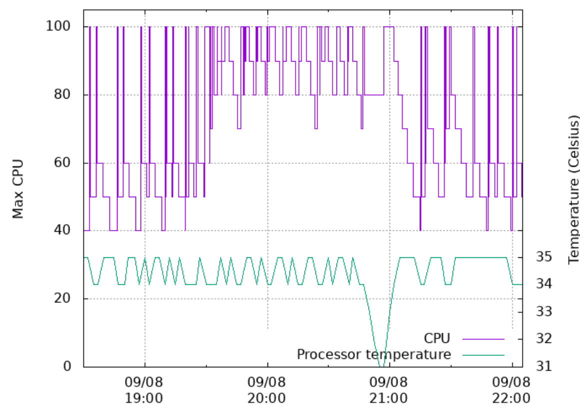



Figure 6: Evolution of the temperature and of the maximum allowed CPU activity.

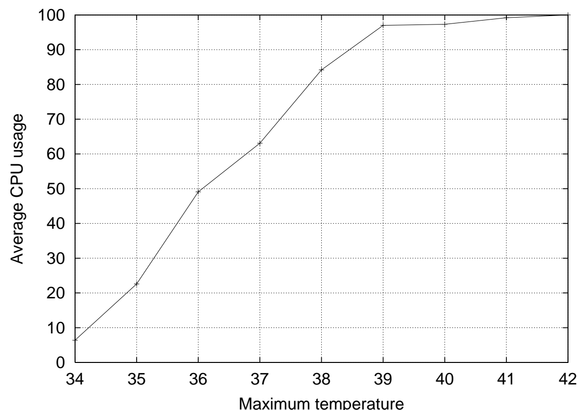



Figure 7: Slowdown of hot processes versus maximum temperature.

impact when it applies to an already low threshold. Conversely, lowering the threshold from 42°C to 39°C reduces the performance by only 3 %. This is significant, since we are interested in controlling the processor temperature when it approaches the higher limit. Doing so in software allows us to maintain a safe temperature at the expense of only a small loss in performance.

## 5 Conclusion

In this paper we present an adaptive approach for controlling both the power consumption and the temperature of a processor. It consists of letting the processor run at full speed while monitoring the temperature. When the temperature reaches a predefined threshold, the activity of processor is selectively reduced so that the constraints can be met. This feedback is necessary since the temperature of the processor depends on various parameters, including the temperature of the room which cannot be predicted from only system statistics.

Our technique is implemented in the operating system, which can both access hardware statistics and control the interleaving of running processes. The system regularly samples the temperature of the processor. If this temperature reaches a predefined threshold, hot processes are identified and slowed down.

Our approach has the advantage of penalizing one application at a time, as opposed to hardware techniques such as throttling which penalize all applications. Consequently interactive applications still run at full speed and their response time is not degraded. It is also superior in that it can make fine adjusts in the CPU activity based on the workload, while throttling can only achieve a few fixed fractions of the maximum CPU activity (usually 50 %, or 33 % and 66 %). Moreover, near the temperature maximum, our results show that we can significantly reduce the temperature with little cost in performance.

## 6 Acknowledgments

Erven Rohou was supported in part by a DARPA grant no. NDA904-97-C-0225. Michael D. Smith is funded in part by a NSF Young Investigator award (grant no. CCR-9457779), NSF grant no. CDA-94-01024, DARPA grant no. NDA904-97-C-0225, and research grants from AMD, Compaq, Digital Equipment, HP, IBM, Intel, and Microsoft.

## References

[1] D. Albonesi. Dynamic IPC/Clock rate optimization. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA-98)*, volume 26,3, pages 282–292, New York, June 27–July 1 1998. ACM Press.

[2] M. Beck et al. *Linux kernel internals*. Addison-Wesley, Reading, MA, USA, 1998.

[3] J.-M. Chang and M. Pedram. Register allocation and binding for low power. In *Proceedings of the 32nd ACM/IEEE conference on Design automation conference*, pages 29–35, 1995.

[4] Intel Corp., editor. *Pentium Pro Family Developer's Manual*. McGraw-Hill Companies, 1999.

[5] Winbond Electronics Corp. *W83781D Winbond H/W Monitoring IC*, November 1997.

[6] D. Cullers, I. Linscott, and B. Oliver. Signal processing in SETI. *Communications of the ACM*, 28(11):1151–1163, November 1985.

[7] E. Feigin. A case for automatic run-time code optimization. Bachelor of Arts Thesis, Harvard College, April 1999.

[8] C. Gebotys. Low power memory and register allocation using network flow. In *DAC*, pages 435–440, 1997.

[9] Intel. *Pentium III Xeon Processor SMBus Thermal Reference*, March 1999. Application Note 914.

[10] Intel. Private communication, August 1999.

[11] Intel, Microsoft, and Toshiba. *Advanced Configuration and Power Interface Specification*, February 1999.

[12] M. Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and minimization techniques for embedded DSP software. *IEEE Transactions on VLSI Systems*, March 1997.

[13] T. Lee and V. Tiwari. A memory allocation technique for low-energy embedded DSP software. In *IEEE Symposium on Low Power Electronics*, San Diego, CA, October 1995.

[14] D. Liu and C. Svensson. Trading speed for low power by choice of supply and threshold voltages. *IEEE Journal of Solid-State Circuits*, SC-28(1):10–17, January 1993.

[15] H. Sanchez, R. Philip, J. Alvarez, and G. Gerosa. A CMOS temperature sensor for PowerPC RISC microprocessors. In *Symposium on VLSI Circuits*, pages 13–14, 1997.

[16] W. Stevens. *TCP/IP Illustrated - The Protocols*, chapter 21, pages 299–322. Addison-Wesley, Reading, MA, USA, 1994.

[17] C.-L. Su, C.-Y. Tsui, and A. Despain. Low power architecture design and compilation techniques for high-performance processors. In *COMPCON*, pages 489–498, San Francisco, CA, March 1994. Also technical report ACAL-TR-94-01 from Univ. of Southern California.

[18] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *IEEE Symposium on Low Power Electronics*, San Diego, CA, October 1994.

[19] M. Toburen, T. Conte, and M. Reilly. Instruction scheduling for low power dissipation in high performance microprocessors. Technical report, North Carolina State University, May 1998.

[20] N. Zervas, K. Masselos, and C. Goutis. Code transformations for embedded multimedia applications: Impact of power and performance. Power-Driven Microarchitecture Workshop In Conjunction With ISCA98, June 1998.