HAL
open science

Scheduling issues on thermally-constrained processors

Pierre Michaud, Yiannakis Sazeides

To cite this version:

Pierre Michaud, Yiannakis Sazeides. Scheduling issues on thermally-constrained processors. [Research Report] PI 1822, 2006, pp.19. inria-00110507

HAL Id: inria-00110507

https://inria.hal.science/inria-00110507

Submitted on 30 Oct 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

I R I S A

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

P U B L I C A T I O N
I N T E R N E
N° 1822

SCHEDULING ISSUES ON THERMALLY-CONSTRAINED PROCESSORS

PIERRE MICHAUD , YIANNAKIS SAZEIDES

ISSN 1166-8687

I R I S A

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Scheduling issues on thermally-constrained processors

Pierre Michaud *, Yiannakis Sazeides **

Systèmes communicants
Projet CAPS

Publication interne n° 1822 — Octobre 2006 — 19 pages

Abstract: The relentless increase of power density has rendered temperature a primary design constraint for microprocessors. Although a power density increase does not necessarily lead to a higher time-average temperature, we show in this study that it increases the amplitude of temperature oscillations. As a consequence, thermal throttling may be engaged more often and degrade performance. We establish that a possible way to decrease the amplitude of temperature oscillations is to increase their frequency. In a multiprogrammed environment, executing multiple threads/processes alternately can result in temperature oscillations if different threads generate different power densities. We therefore suggest that, as power density increases and processors get faster, the time slice can and should be decreased. Using a short time slice also enables the operating system to take advantage of activity migration without special architectural support. Indeed, on a thermally constrained multi-core (TCMC), activity migration can be leveraged to decrease temperature by better distributing heat over the different cores. Furthermore, we show that fair scheduling with different thread priorities can be achieved but requires sometimes to run simultaneously fewer threads than cores when fewer threads are sufficient to maintain the TCMC at thermal saturation. We propose a scheduling method that implements activity migration while taking into consideration different thread priorities. We show that, under a temperature constraint, this scheduling method provides a fair partitioning of the overall computing power of a TCMC while delivering a global execution throughput close to the maximum throughput.

Key-words: Multi-core processor, temperature, thread scheduling, time slice, activity migration

(Résumé : tsvp)

* pmichaud@irisa.fr
** yanos@cs.ucy.ac.cy

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Ordonnancement de processus sur processor sous contrainte thermique

Résumé : De par l'augmentation continue de la densité de puissance dissipée, la température est devenue une contrainte importante dans la conception des microprocesseurs. Bien qu'une augmentation de la densité de puissance n'augmente pas nécessairement la moyenne temporelle de la température, nous montrons dans cette étude que cela augmente l'amplitude des oscillations de température. En conséquence, les mécanismes automatiques de réduction de la puissance dissipée, qui diminuent la performance, se déclenchent plus souvent. Nous montrons qu'augmenter la fréquence des oscillations est un des moyens permettant de réduire leur amplitude. Dans un environnement multiprogrammé, l'exécution alternée de plusieurs processus peut générer des oscillations de température si les différents processus produisent différentes densités de puissance. Nous en déduisons que l'augmentation de la densité de puissance et de la performance des processeurs doit et peut s'accompagner d'une diminution du quantum de temps. De plus, un petit quantum de temps permet au système d'exploitation de bénéficier de la migration d'activité sans matériel spécial. En effet, sur un processeur multi-coeur sous contrainte thermique (TCMC), la migration d'activité permet de diminuer la température en répartissant la dissipation de chaleur également sur les différents coeurs. De plus, nous montrons qu'un ordonnancement équitable avec différentes priorités de processus nécessite parfois d'exécuter simultanément moins de processus que de coeurs lorsque cela est suffisant pour maintenir le TCMC en saturation thermique. Nous proposons une méthode d'ordonnancement qui utilise la migration d'activité en prenant en compte les priorités de processus. Nous montrons que, sous contrainte thermique, cette méthode d'ordonnancement répartit la puissance de calcul d'un TCMC équitablement tout en délivrant un débit d'exécution global proche du débit maximum.

Mots clés : Processeur multi-coeur, température, ordonnancement de processus, quantum de temps, migration d'activité

1 Introduction

Temperature has become an important design constraint for microprocessors. The temperature problem is mainly a consequence of circuit miniaturization and constrained voltage scaling. Temperature on a chip is neither uniform nor constant. In particular, it depends on applications characteristics, on the number of running threads [1], and on the room temperature. Current processors feature thermal throttling mechanisms that prevent temperature from exceeding a fixed value [6, 4]. While thermal throttling is designed as an emergency mechanism on current processors, it will become more and more a normal event in future thermally constrained processors, especially server-class multi-cores which are designed for maximum throughput.

The temperature constraint has several implications on thread scheduling. The goal of this study is to explore some of these implications. In Section 2, we show that the thermal design of thermally constrained processors should not be solely based on steady-state temperature, but also on transient temperature. In particular, it is important to take into account temperature oscillations due do thread scheduling. We show that the amplitude of temperature oscillations increases with power density, and that a possible way to counteract this effect is to increase the oscillation frequency. This leads us to argue in favor of small OS time slices. Section 3 studies the impact of power density and time slice on the performance of a thermally constrained processor. Section 4 focuses on thermally-constrained multi-cores. When activity migration is used [7, 14], multi-cores offer a way to fight the temperature problem. In this case, temperature oscillations mostly come from cores being alternately active and inactive. Previous work about activity migration has considered threads with equal priority. Section 4 addresses the question of how to take advantage of activity migration when threads have different priorities. We show that fair scheduling requires, sometimes, to run fewer threads than cores. We propose a scheduling method, based on short time slices, that achieves fairness while maintaining cores at thermal saturation. We also propose an alternative method where thread migration is managed by the firmware on the basis of thermal sensor information, the OS being responsible only for deciding which thread to run at a given time.

1.1 Methodology

All results in this study are based on simplified models. We model the power density map with square power sources. Power numbers are synthetic. All temperature numbers are obtained with ATMI [1]. The ATMI physical model is depicted on Figure 1, where layer 1 represents the silicon die and layer 2 the copper heat-sink base-plate. The parameter values used in this study are given in Table 1. The expression *relative temperature* used in following sections denotes the temperature rise above the ambient temperature, which is the temperature of the air hitting the heat-sink (i.e., inside the computer box).

[1] In this study, we do not distinguish between threads and processes.

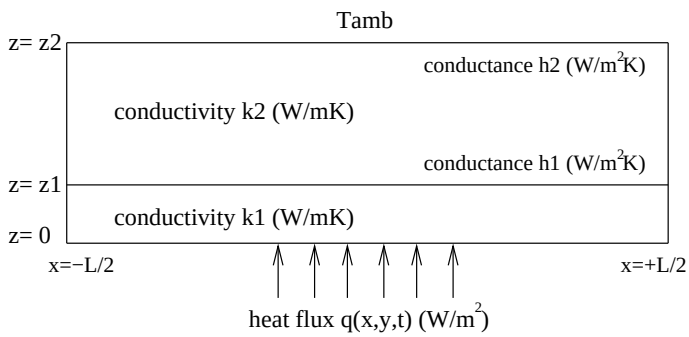

Figure 1: *ATMI temperature model*

parameter	value	unit	physical meaning
k_1	110	W/mK	silicon thermal conductivity
k_2	400	W/mK	copper thermal conductivity
α_1	6×10^{-5}	m^2/s	silicon thermal diffusivity
α_2	1.1×10^{-4}	m^2/s	copper thermal diffusivity
h_1	3×10^4	W/m^2K	interface material thermal conductance
h_2	700	W/m^2K	heat-sink thermal conductance
z_1	0.5	mm	die thickness
$z_2 - z_1$	5	mm	heat-sink base plate thickness
L	7	cm	heat-sink base plate width

Table 1: *ATMI parameters.*

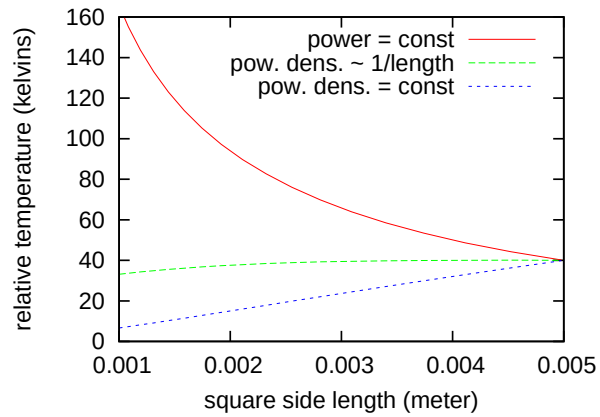

Figure 2: *Steady-state relative temperature as a function of dimension, starting (on the right side) from a* $5mm \times 5mm$ *square source dissipating 25 watts, and scaling it down to* $1mm \times 1mm$ *(on the left side) according to different scaling scenarios : constant power, constant power density and power density increasing as the inverse of length*

2 The Temperature Problem

The increase of power density is often mentioned to explain why temperature is becoming a problem on high performance chips. Power density is indeed an important parameter. But when power density increases because circuits size is reduced, this does not necessarily imply a higher steady-state temperature. On the other hand, a higher power density increases the amplitude of temperature oscillations. We believe one cannot fully understand the temperature problem and ways to tackle it without understanding the issue of temperature oscillations. In this section we discuss the temperature problem and how it can be mitigated by investigating the effects of power density on temperature.

2.1 Steady State

Figure 2 shows the result of taking a $5mm \times 5mm$ square source dissipating 25 watts uniformly and scaling its dimensions down to $1mm \times 1mm$, according to three different scenarios : power is kept constant, power density is kept constant, or power density increases as the inverse of the square side length. This example shows the impact of circuit miniaturization. As illustrated in the graph, keeping total power constant, but concentrating it in an area that gets smaller and smaller, leads to a dramatic increase of steady-state temperature. On the other hand, keeping power density constant brings temperature down. The middle curve is for a power density that is kept proportional to the inverse of length. This results in temperature being roughly constant, only slightly decreasing with length. Hence the region

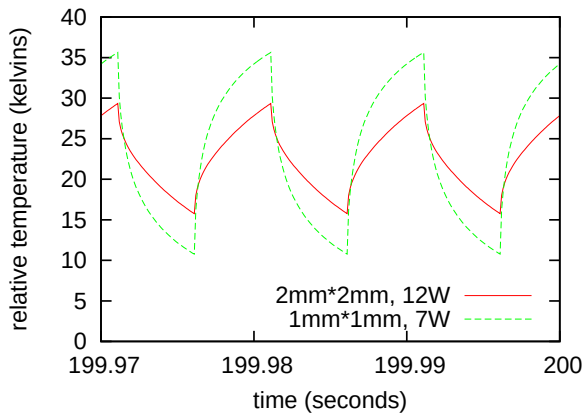

Figure 3: *Relative temperature as a function time for a square power source being periodically on and off. One curve is for a* $2mm \times 2mm$ *square generating 12 watts when the source is on, the other curve for a* $1mm \times 1mm$ *square generating 7 watts when the source is on.*

of the graph lying between the two dashed curves is a region of higher power density, yet lower temperature. This shows that an increase of power density does not necessarily mean a higher temperature. In this particular example, when we scale down the square length by a factor $\lambda < 1$, steady-state temperature does not increase as long as power density is not increased by more than $\lambda^{-1.1}$ (approximately). If power density increases faster than that, steady-state temperature may become a problem, unless the packaging is improved. This reasoning is for a single circuit whose dimensions are scaled down. However, if one takes advantage of miniaturization to put more circuits in a given area, the temperature problem becomes worse.

2.2 Temperature Oscillations

The power density in a region of the chip depends on applications runtime characteristics. For example, CPU-bound applications generate a high power density in the execution core, while memory-bound applications generate a lower power density. In a multiprogrammed environment, where different applications execute alternately, these disparities generate temporal power density variations, hence temporal temperature variations.

Figure 3 shows temperature given by ATMI as a function of time for a square power source being periodically on and off. One curve is for a $2mm \times 2mm$ square generating 12 watts when the source is on, the other curve for a $1mm \times 1mm$ square generating 7 watts when the source is on. As expected, a periodic power density oscillation generates a temperature oscillation with the same period. So the period is the same in both cases. However, the amplitude of the temperature oscillation is more pronounced for the small

source, because of the higher power density ($7\,W/mm^2$ vs. $12/4 = 3\,W/mm^2$), despite both curves having approximately the same time-average value.

One can reason about temperature oscillations using the following rule of thumb [12] :

> *The amplitude of a temperature oscillation is proportional to the local power density swing and inversely proportional to the square root of the oscillation frequency.*

This rule of thumb is valid for high oscillation frequencies only (i.e., for a period not exceeding a millisecond). Nevertheless, it highlights the fact that local power density is the parameter whose impact on temperature oscillations is the most significant, and that a possible way to decrease the amplitude of temperature oscillations is to increase the oscillation frequency.

2.3 Thermal Throttling

In order to keep temperature below the thermal limit T_{max}, modern processors feature thermal throttling mechanisms that reduce power consumption when temperature approaches T_{max}. For instance, the Intel Pentium 4 uses an on-off mechanism that stops the clock for several microseconds whenever the on-chip thermal sensor indicates that the thermal limit T_{max} is exceeded [6]. In the remaining, we assume an on-off thermal throttling mechanism with a duty cycle which adjusts automatically to the maximum possible value. This is achieved simply by stopping the clock for a fixed off-time whenever temperature reaches T_{max}.

During the off-time, dynamic power consumption is greatly reduced. As for static power in logic circuits, it can be decreased by using high-V_t sleep transistors [17]. In the remaining, we idealize this situation by assuming a null core power consumption during off times (actually, some static power is still dissipated in caches and other microarchitected tables).

On-off thermal throttling generates a temperature oscillation. If we want the amplitude of this oscillation to be small compared to $T_{max} - T_{amb}$ (so that time-average temperature is as close to T_{max} as possible), the off-time should be small enough. According to the rule of thumb for temperature oscillations, when power density doubles, the off-time should be taken 4 times shorter. In remaining experiments, we assume an off-time of $2\,\mu s$ and a maximum temperature $T_{max} = 85°\mathrm{C}$.

3 A Case for Short Time Slices

When two threads are executed alternately on a processor, a power density oscillation may be generated at some points of the chip, with a period equal to the sum of time slices of each thread. This occurs typically when one thread uses a unit that the other thread uses infrequently, e.g., the floating-point unit, or when thread characteristics give more opportunities for clock gating (e.g., one thread experiences a lot of cache misses). Figure 4 shows an abstract view of a processor core, which we represent as a $5\,mm \times 5\,mm$ square power source dissipating $1\,W/mm^2$. Let us consider a unit, which we represent as a small

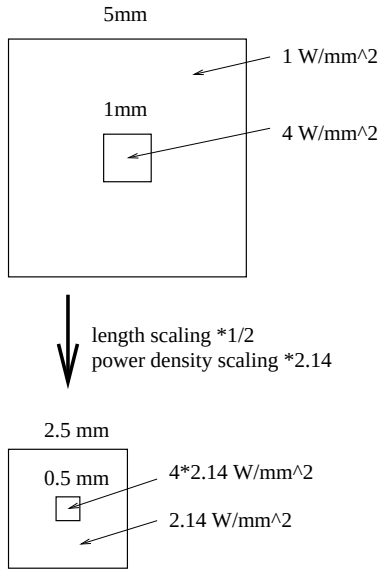

Figure 4: *Abstract model of processor core.*

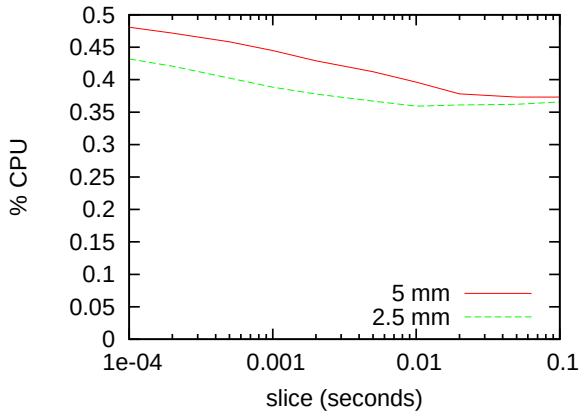

Figure 5: *Fraction of CPU time for thread #1 as a function of the time slice, for both cases depicted on Figure 4.*

$1\,mm \times 1\,mm$ square, that is used by thread #1 and not used by thread #2. When the unit is used, we assume it dissipates $4\,W/mm^2$. We assume that a thermal sensor is located at the center of the unit. We assume $T_{amb} = 40$ °C, that is, the maximum relative temperature is 45 °C. When temperature in the unit exceeds $T_{max} = 85$°C, we stop the clock for $2\,\mu s$ and assume a null power consumption in the meantime. We assume that both threads have the same priority and are not preempted. So each thread has the same fixed time slice. When the thread currently running has exhausted its time slice, the OS runs the other thread, and so on. As a measure of performance, we give the fraction of CPU time used by each thread, not counting in this fraction the time lost because of thermal throttling. [2] For the initial thermal state, we assume that for $t < 0$ thread #1 has been running alone for a long time, i.e., the unit is thermally saturated. Then for $t \geq 0$, threads #1 and #2 are executed alternately. Figure 5 shows the fraction of CPU time used by thread #1 as a function of the time slice, after 1 simulated second. The second curve on figure 5 is for a processor whose dimensions are scaled down by a factor $\lambda = 1/2$ and power density scaled up by a factor $\lambda^{-1.1} \approx 2.14$ (cf. Section 2.1), as shown on Figure 4.

Figure 5 shows that the performance of thread #1 decreases significantly when the time slice increases (on this example, more than 20% performance loss compared with the maximum 50% CPU fraction). When circuits dimensions are scaled down and power density is increased, the problem is even worse. Figure 6 shows a snapshot of temperature as a function of time for a time slice of $1\,ms$ and $10\,ms$. When thread #2 is running, temperature drops because no power is dissipated in the unit. When thread #1 is running, temperature increases until the maximum temperature is reached. Then thread #1 is throttled. As can be seen on Figure 6, taking a shorter time slice, decreases the amplitude of the temperature oscillation, and allows time-average temperature to be closer to T_{max}. This illustrates how a shorter time slice can increase performance on thermally constrained processors.

The main insight from this example is that, if power density on future processors keeps rising with technology shrinking, taking shorter time slices may lessen the temperature constraint, especially in server-class processors. Of course, short time slices incur some cache penalty, and this has to be taken into account. We ran Simplescalar out-of-order microarchitecture model [2] with a 512-Kbyte level-2 cache and with a stride prefetcher. We measure the IPC and we assume a 5 Ghz clock. We considered only 2 cache levels. Upon a level-2 cache miss, the missing block is retrieved from main memory, with an initial latency of 150 CPU cycles. To get an idea of the cache penalty incurred by a short time slice, we flush all caches every 1 millisecond, that is, every 5 millions cycles. The worst performance loss observed on the SPEC CPU 2000 is about 8%, with an average performance loss of about 3% (we get slightly higher penalties when we disable the stride prefetcher). It should be noted that these are worst-case numbers. In reality caches are not flushed. The working sets of the different threads share the cache capacity. If caches are big enough to hold all

[2] The time lost because of thermal throttling is easily defined here. It is the total time spent with the clock stopped. If one uses dynamic voltage scaling, the time lost is more difficult to measure, but could be defined as the time we would gain if temperature were not limited.

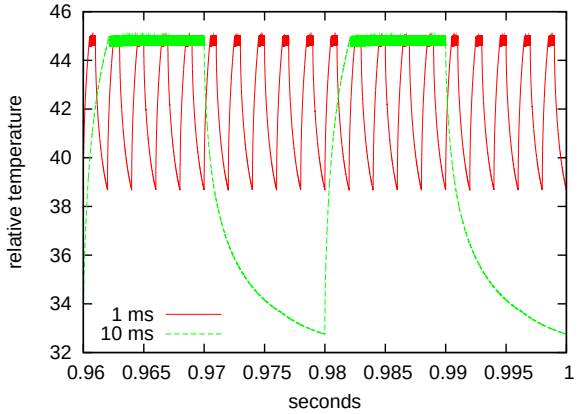

Figure 6: *Snapshot of temperature as a function of time for two different time slices.*

working sets without conflicts, then the penalty from taking a short time slice should be negligible compared with the potential performance gains under a strong thermal constraint.

Nevertheless, it may become necessary, in future processors, to consider time slices much shorter than a millisecond, as illustrated on Figure 5. In that case, large on-chip caches and hardware support for fast context switches may be the only solution to tolerate short time slices. Though large L2 and L3 caches do contribute to the overall static power consumption, they have a relatively low power density and are not the hottest regions of the chip. Consequently, we believe that large on-chip caches will become an indirect way to fight the temperature problem.

In the remaining of this study, we neglect the context switch penalty.

3.1 Smart Scheduling ?

A short time slice is not the only way to increase the frequency of temperature oscillations. When more than two threads are using the same core in time-sharing fashion, it is possible to take advantage of threads characteristics. For example, let us consider four threads running on a core similar to that modeled on Figure 4, and such that threads 1 and 2 use the hot unit while threads 3 and 4 do not. If the OS schedules threads in the order 1,3,2,4 repeatedly, the temperature oscillation frequency is twice higher than if the schedule is 1,2,3,4. To take advantage of this, the OS should have a knowledge of threads behavior. For example, the OS could record for each thread and for each thermal sensor whether the thread is hot or cold, as in [8]. Then, threads could be ordered so as to alternate hot and cold threads for each sensor. However, though potentially interesting in some situations, the extent to which smart thread scheduling can increase the frequency of temperature oscillations seems limited compared with taking a shorter time slice.

4 Scheduling on a Thermally-Constrained Multi-Core

The previous discussion concerns both single-core and multi-core processors. In the case of a multi-core, the possibility to migrate a thread to a different core brings an extra dimension to the scheduling problem.

Putting multiple cores on a chip offers a way to offset the increase of power density, thanks to *activity migration*. The idea is that while a core is running a "hot" thread, other cores may be inactive or running "cold" threads. By periodically migrating a thread's execution to a different core [7], cores are used evenly and the time-average power density gets divided by the number of cores a thread is using. This technique has been shown to provide significant performance gains under a strong thermal constraint [14]. So the advent of multi-cores, which is partly due to the difficulty of pushing superscalar microarchitectures further, is also motivated by the temperature problem.

Previous work on activity migration has mostly studied its performance efficiency assuming threads with equal priority [7, 14]. When threads have equal priority, the overall throughput of a TCMC can be maximized by running simultaneously as many threads as possible (as long as shared caches can hold the working sets). In this context, previous work has studied questions like which thread to run, and on which core.

However, when threads have different priorities, we show in this section that it may be necessary sometimes to run fewer threads than cores. In this case, temperature oscillations come from cores being successively active and inactive. Such temperature oscillations have a wider amplitude than those generated when alternating threads with different characteristics, because power density swings over larger areas. Hence the problem of temperature oscillations is essential on a TCMC.

The effect of having a mix of threads with different characteristics, which activity migration can exploit, has been extensively studied in previous work [14]. In this study, our conclusions are essentially orthogonal to previous works, and we consider threads with identical thermal characteristics in order to simplify the study. In the remaining, we consider a TCMC where each core has an independent throttling mechanism, i.e., only the cores whose thermal sensors indicate a temperature exceeding T_{max} have their power consumption throttled. This way, cores that are below the temperature limit can continue working normally.

4.1 Time Slice Definition.

We must first clarify the notion of time slice on a TCMC. In the example of Section 3, we used the usual definition, i.e., the time-slice is a "real-time" slice (RTS), meaning that at each timer interrupt, we remove one timer period from the remaining time slice of the thread that was running during that period. For example, if the time slice equals 10 timer periods, we consider the time slice exhausted after 10 timer interrupts, no matter how long the clock was stopped because of thermal throttling. But we could also define an active-time slice (ATS), such that at each timer interrupt, we subtract from the remaining time slice the time spent with the clock on, i.e., not counting the time lost because of thermal throttling. In

this case, for a fixed ATS, the more the thread is throttled the longer the RTS. One sees immediately that using ATS is not viable, as it allows a hot thread to take up most of the CPU time. Hence in the remaining we take the usual RTS as the time-slice definition.

4.2 Scheduling Dilemma

On a TCMC, the instantaneous per-thread performance is likely to be higher when running fewer threads at the same time. For example, let us consider two cores, and two threads with similar characteristics but different priorities. One thread is a high-priority (HP) thread, and the other thread a low-priority (LP) one, so the OS gives a longer time slice to the HP thread. If we keep running the HP and LP threads simultaneously on the two cores, we are giving the same CPU time to each thread, despite them having different priorities (when the time slice for a thread is exhausted, it is renewed right away as we consider only two threads in this example). However, if thermal throttling triggers when executing both threads simultaneously, we know that we can increase the performance of the HP thread by allowing the HP thread to run alone, taking advantage of activity migration [7, 14]. This may be considered fairer for the HP thread. So we could choose to run the HP and LP threads alternately, as if the dual-core were a single-core, trying to use both cores evenly to minimize the peak temperature. However if we do this, we may waste processor performance. In other words, running two threads simultaneously may be sufficient to keep cores thermally saturated, while running a single thread may not. Therefore, in order to best exploit a TCMC, the OS should take into account the thermal state of the whole multi-core.

4.3 Example Scheduling Method

When scheduling threads on a multi-processor, some operating systems like Linux maintain a separate *run queue* of runnable threads for each processor [10]. On a TCMC, a single run queue seems more natural, as cores influence each other through global heating. Henceforth, we assume a single run queue. In Linux, the CPU time is divided into *epochs*. At the beginning of each epoch, each (non real-time) thread is given a time slice that is computed from its base priority (and for IO-bound processes, from the unused time slice from the previous epoch) [3]. For non real-time threads, the higher the priority, the longer the time slice. At the beginning of an epoch, the scheduler selects from the run queue the thread with the largest slice and executes it. The time slice for that thread is decremented at each timer interrupt. Unless the thread is preempted by another thread with higher priority, or voluntarily relinquishes the CPU (e.g., because it blocks for an I/O), the thread executes until its time slice is exhausted. After the time slice is exhausted, the thread is no longer runnable for the rest of that epoch. Then the scheduler selects the next runnable thread with the largest slice, and so on. When all runnable threads have exhausted their slice, the current epoch ends, and a new epoch begins : threads are given a new time slice, and the cycle repeats.

To illustrate the possibility of temperature-aware scheduling with different thread priorities, we did the following experiment. We assume a 4-core processor, where each core is

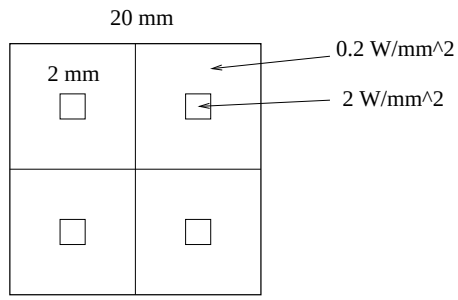

Figure 7: *4-core processor model*

modeled as a $1\,cm \times 1\,cm$ square, as depicted on Figure 7. We consider identical threads such that, when executed on a core, the core dissipate $0.2\,W/mm^2$ except in a small $2\,mm \times 2\,mm$ square where power density is $2\,W/mm^2$. We assume that there is a thermal sensor at the center of each core and that each core has an independent thermal throttling mechanism. The maximum temperature is $T_{max} = 85$°C. We set the initial thermal state as if each core had been running a thread for a long time. We propose and evaluate the following scheduling method. We assume a timer period of $1\,ms$, equal to the scheduling period. We assume that the OS can read all thermal sensors, and that there is a counter giving the number of times thermal throttling has been triggered since the last timer interrupt. Every millisecond, we count the number N of runnable threads, i.e., threads that have not exhausted their time slice. If $N = 0$, we renew all slices and start a new epoch. Otherwise if $1 \leq N < 4$ (either because of a small load or because some threads have exhausted their time slice), we compute the average temperature, i.e., the sum of temperatures on the 4 cores, divided by 4. We define a thermal saturation temperature $T_{sat} = T_{max} - 3$. There are two cases to consider :

- **Thermal saturation** : the average temperature exceeds T_{sat} and thermal throttling triggered more than once since the last timer interrupt. In this case, we run only the N runnable threads for the next timer period, that is, $4 - N$ cores are not used.
- **Under-saturation** : the average temperature is below T_{sat} or thermal throttling triggered at most once. We select $4-N$ threads among the threads that have exhausted their time slice, and we give them a time slice of one timer period.

Then we remove threads from cores, select new threads with the largest slices, and map these selected threads on cores in a random fashion. It should be noted that the new threads are not necessarily different from the ones that were running during the previous timer period. But they have a certain probability to run on different cores. This form of activity migration equalizes time-average temperature on the different cores, so that thermal throttling triggers as infrequently as possible. We do not claim this scheduling method to be the best possible

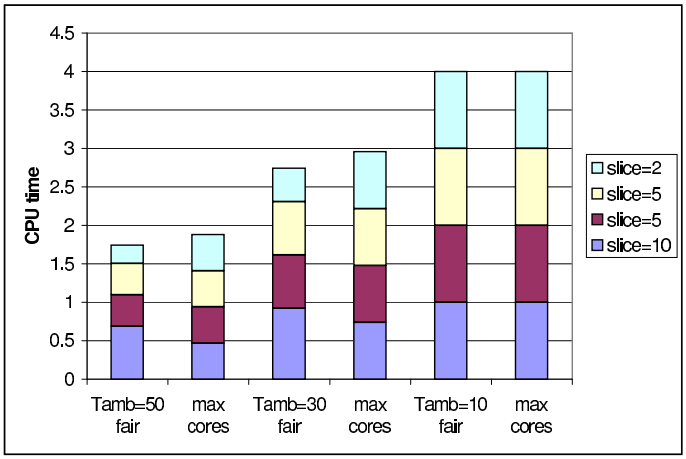

Figure 8: *Fraction of CPU time for a load of 4 threads on a 4-core processor (maximum performance is* $4 \times 100\% = 4$*). Threads have different priorities : one thread has a time slice of 10 timer periods, two threads have a slice of 5, and the fourth thread a slice of 2 timer period. From left to right, the ambient temperature is 50°C, 30°C, and 10°C respectively. For a given ambient temperature, we show two bars : one for the fair scheduling method ("fair"), and the other for the usual method that keeps executing a thread on each core ("max cores").*

one. However, this method has two properties that we believe a good scheduling method for TCMCs should have :

- it maintains all cores close to thermal saturation,
- sometimes it runs fewer threads than cores when threads have different priorities.

Figure 8 shows the fraction of CPU time for a load of 4 threads and for various ambient temperatures. One thread has a time slice of 10 timer periods, two threads have a slice of 5, and the fourth thread a slice of 2 timer periods. This graph compares the scheduling method introduced above ("fair") and the trivial scheduling method consisting of always executing a thread on each core ("max cores").

As can be observed, the fair scheduling method decreases slightly the total performance, but it gives more performance to the high-priority thread. For instance, at $T_{amb} = 30$°C, the total performance loss is 7%, but the highest-priority thread gets 25% more performance with fair scheduling. In this example, it takes a very low ambient temperature (10°C) for the multi-core to run 4 threads simultaneously without being throttled. Full performance can only be obtained in an air-conditioned data center and provided the inlet air is not heated before reaching the processor heat sink. In this case, each thread has 100% CPU time, for

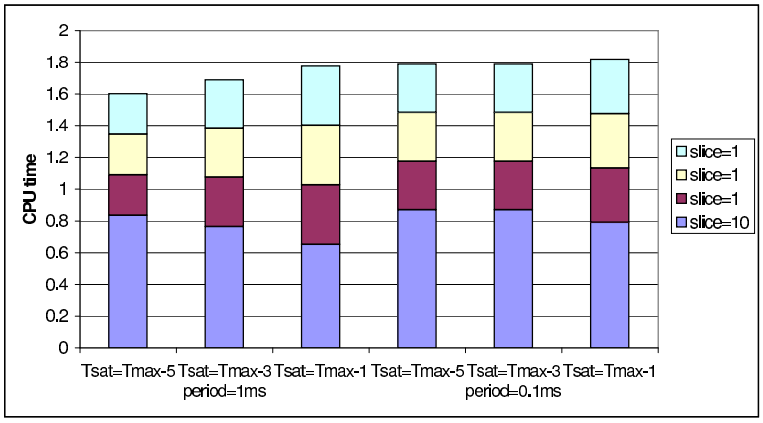

Figure 9: *Fraction of CPU time for 4 threads (slice=10,1,1,1) and for various values of* T_{sat} *at* $T_{amb} = 50\,°C$*. The 3 bars on the left side are for a timer period of* $1\,ms$*, the 3 bars on the right side are for a timer period of* $100\,\mu s$*.*

a total performance of 4. When the ambient temperature is 30°C, The total performance is less than 4, but the performance of the highest-priority thread is greater than 90%.

When the ambient temperature is 50°C, the highest-priority thread is throttled but still has a larger portion of the overall performance. The total performance is about 1.7, which is less than the performance of 2 cores when there is no thermal constraint. However, if we use only two cores while turning off the two other cores, the total performance for the same 4 threads and at the same 50°C ambient temperature is only 1.25. This example illustrates the advantage of having multiple cores with respect to thermal throttling : by using 4 cores instead of 2, the time-average power density is halved.

4.3.1 Impact of T_{sat} value

Results of Figure 8 are for $T_{sat} = T_{max} - 3K$ and a timer period of $1\,ms$. Figure 9 shows the CPU time for different values of T_{sat} (5, 3 and 1 kelvin below T_{max}) and for a timer period of $1\,ms$ and $100\,\mu s$, at $T_{amb} = 50$°C. One thread has a slice of 10 timer periods, the 3 other threads a slice of 1. As expected, total performance decreases with T_{sat}, because we increase the amplitude of temperature oscillations. So T_{sat} should not be too small. On the other hand, if we take T_{sat} too close to T_{max}, the scheduling algorithm often runs the maximum number of threads because it cannot distinguish between under-saturation and the unavoidable temperature drop on unused cores in one schedule period. Hence as T_{sat} approaches T_{max}, the performance of the high priority thread decreases. For example, with a timer period of $1\,ms$, we get 10% total performance loss when going from $T_{sat} = T_{max} - 1K$ to $T_{sat} = T_{max} - 5K$, but a 28% performance gain for the high-priority thread. A good value

for T_{sat} is the result of a trade-off. It depends on power density and on the schedule period. A higher power density or a longer schedule period means larger temperature oscillations, which requires a smaller T_{sat} value. For example, as shown on Figure 9, taking a schedule period of $100\,\mu s$ instead of $1\,ms$ permits ensuring fairness with a higher T_{sat} hence a greater overall performance.

4.4 Sensor-Triggered Migrations

In the scheduling method of Section 4.3, the OS is responsible for deciding which thread to run at a given time, and on which core. We ensured activity migration by mapping threads on cores in a random fashion every millisecond. Statistically, each thread visits all cores, and cores have roughly the same time-average temperature. However, this method is viable only if it does not induce too many cache misses. If power density keeps increasing with technology shrinking, temperature oscillations due to cores being alternately active and inactive may require a schedule period much shorter than a millisecond. In this case, we may want to keep executing a thread on the same core as long as possible.

We propose a variant of the previous scheduling method in which the OS is no longer responsible for mapping threads on cores. The OS chooses which thread should run at a given time and, we assume, some firmware is responsible for activity migration. To prevent unnecessary migrations, we take advantage of the following observation. We need to migrate only when we detect a cold core ($T < T_{sat}$) **and** there exists a running thread which is hot (thermal throttling triggered recently). If such occurrence is detected, we move the hot thread to the cold core. If there was a thread running on the cold core, we move it to the hot core. That is, we exchange threads. [3] To prevent threads from migrating too frequently, a thread is not candidate for migration before a certain time t_{mig} has elapsed since its last migration. The value of t_{mig} should be small enough for limiting the amplitude of temperature oscillations, yet not too small to limit induced cache misses. Figure 10 compares random mapping and sensor-triggered migrations at $T_{amb} = 30$°C. We give for each thread the average time between context switches. We count a context switch for a thread each time the thread is put on a core and the previous thread running on that core was a different one. For sensor-triggered migrations, we set $t_{mig} = 1\,ms$, and we consider that a thread is hot if it triggered thermal throttling during the last $20\mu s$. For both methods, we assume a $1\,ms$ schedule period. We verified that both methods give approximately the same performance if we neglect induced cache misses, with sensor-triggered migrations giving a slightly higher performance (on this example, +3% overall and +2% for the highest-priority thread) The main difference is that sensor triggered migrations permit decreasing the number of context switches. As can be seen on Figure 10, this benefits mostly to the high-priority thread, as this thread is the one that uses activity migration most often.

[3] In this paper, for simplifying the discussion, we assume a single thermal sensor per core. If there are several sensors, we need to migrate a thread when detecting a cold sensor **and** another core on which the same corresponding sensor triggered thermal throttling recently.

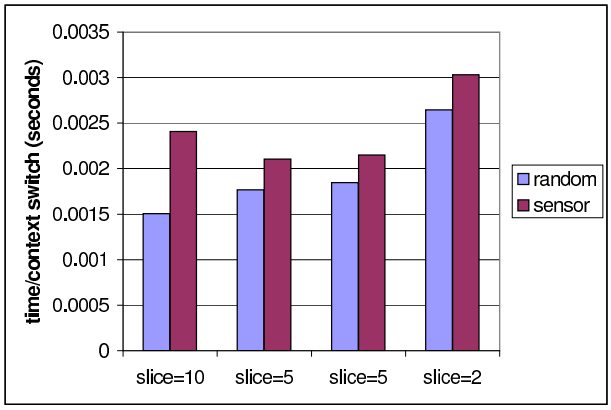

Figure 10: *Average time (in seconds) between context switches at $T_{amb} = 30\,^{\circ}C$. Comparison between random core mapping and sensor-triggered migrations.*

5 Related Work

The need for short time slices in thermally-constrained processors, even if implicit in a couple of studies [9, 8], has not been stated explicitly before. Nevertheless, there are a few papers on temperature-aware scheduling. In [15], it was proposed to let the OS keep temperature below T_{max} by identifying the "hot" processes and give them less CPU time. However, on today's processors [6, 4, 13], the task of maintaining temperature below T_{max} is mostly the responsibility of the firmware. If a process is too hot, thermal throttling triggers automatically. In [8], authors propose to schedule threads based on their thermal characteristics recorded during previous time slices. In particular, they propose to alternate "hot" and "cold" threads on a given core.

Previous studies have shown that activity migration on multi-cores [7, 14, 16] or on SMP systems [11] is an efficient technique for decreasing temperature. However, these studies looked only at threads with equal priorities. To our knowledge, the problem of thread scheduling on a TCMC when threads have different priorities has not been considered previously.

6 Conclusion

This paper argues that the thermal design of a thermally constrained processor should not be based solely on steady-state temperature, but also on transient temperature. In particular, we established that the amplitude of temperature oscillations increases with power density. We have shown that the amplitude of temperature oscillations decreases if we increase their frequency. This leads us to argue in favor of small OS time slices.

This paper demonstrates that, on a thermally-constrained multi-core, fair scheduling for threads with different priorities requires sometimes to run fewer threads than cores, so that high-priority threads can take advantage of activity migration. In such case, temperature oscillations mostly come from cores being alternately active and inactive. We have proposed a scheduling method based on short time-slices that takes into account different thread priorities. We have shown that minimal hardware support is sufficient, in the form of an indicator of whether cores are thermally saturated or not. Based on this information, the OS is responsible for choosing how many threads to run at a given time.

This study is mostly qualitative, and details of a scheduling method to be implemented on future systems will depend on the specific technological context, in particular packaging characteristics and power densities. Engineers developing a server-class multi-core should try to get an estimation of the maximum power density in the early stages of the design. This will permit estimating the amplitude of temperature oscillations on typical workloads. From such data, it is possible to decide whether the OS time slice can be made short enough to attenuate oscillations, or whether some architectural supports are required. Some potentially useful architectural supports are :

- larger caches to lessen cache misses induced by context switches,
- letting the firmware manage thread migrations for minimizing the OS overhead, as illustrated in Section 4.4,
- supports for fast thread migrations [16, 5].

References

[1] ATMI. http://www.irisa.fr/caps/projects/ATMI.

[2] Simplescalar. http://www.simplescalar.com.

[3] D.P. Bovet and M. Cesati. *Understanding the Linux kernel*. O'Reilly, 2001.

[4] J. Clabes et al. Design and implementation of the POWER5 microprocessor. In *Proceedings of the 41st Design Automation Conference*, 2004.

[5] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec. Performance implications of single thread migration on a chip multi core. *ACM SIGARCH Computer Architecture News*, 33(4):80–91, November 2005.

[6] S.H. Gunther, F. Binns, D.M. Carmean, and J.C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, (Q1), February 2001.

[7] S. Heo, K. Barr, and K. Asanović. Reducing power density through activity migration. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 2003.

[8] E. Kursun, C.-Y. Cher, A. Buyuktosunoglu, and P. Bose. Investigating the effects of task scheduling on thermal behavior. In *Third Workshop on Temperature-Aware Computer Systems*, 2006.

[9] Z. Lu, J. Lach, M.R. Stan, and K. Skadron. Improved thermal management with reliability banking. *IEEE Micro*, 25(6), November 2005.

[10] P. Mackerras, T.S. Mathews, and R.C. Swanberg. Operating system exploitation of the POWER5 system. *IBM Journal of Research and Development*, 49(4/5):533–539, July 2005.

[11] A. Merkel, F. Bellosa, and A. Weissel. Event-driven thermal management in SMP systems. In *Second Workshop on Temperature-Aware Computer Systems*, 2005.

[12] P. Michaud, Y. Sazeides, A. Seznec, T. Constantinous, and D. Fetis. An analytical model of temperature in microprocessors. research report RR-5744, INRIA, November 2005.

[13] C. Poirier, R. McGowen, C. Bostak, and S. Naffziger. Power and temperature control on a 90nm Itanium-family processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, 2005.

[14] M.D. Powell, M. Gomaa, and T.N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[15] E. Rohou and M.D. Smith. Dynamically managing processor temperature and power. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimization*, 1999.

[16] A. Shayesteh, E. Kursun, T. Sherwood, S. Sair, and G. Reinman. Reducing the latency and area cost of core swapping through shared helper engines. In *Proceedings of the International Conference on Computer Design*, 2005.

[17] J.W. Tschanz, S.G. Narendra, Y. Ye, B.A. Bloechel, S. Borkar, and V. De. Dynamic sleep transistor and body bias for active leakage power control of microprocessors. *IEEE Journal of Solid-State Circuits*, 38(11):1838–1845, November 2003.