A Study of Thread Migration in Temperature-Constrained Multicores

PIERRE MICHAUD, ANDRÉ SEZNEC, and DAMIEN FETIS
IRISA/INRIA
and
YIANNAKIS SAZEIDES and THEOFANIS CONSTANTINOU
University of Cyprus

Temperature has become an important constraint in high-performance processors, especially multicores. Thread migration will be essential to exploit the full potential of future thermally constrained multicores. We propose and study a thread migration method that maximizes performance under a temperature constraint, while minimizing the number of migrations and ensuring fairness between threads. We show that thread migration brings important performance gains and that it is most effective during the first tens of seconds following a decrease of the number of running threads.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Parallel Architectures

General Terms: Performance, Reliability, Management

Additional Key Words and Phrases: Multicore processor, power density, temperature, thermal management, thread migration

ACM Reference Format:

Michaud, P., Seznec, A., Fetis, D., Sazeides, Y., and Constantinou, T. 2007. A study of thread migration in temperature-constrained multicores. Architec. Code Optim. 4, 2, Article 9 (June 2007), 28 pages. DOI = 10.1145/1250727.1250729 http://doi.acm.org/10.1145/1250727.1250729

1. INTRODUCTION

Temperature has become an important constraint for current high-performance processors because of its detrimental effect on circuit timing, mean time to failure, and leakage currents. High temperature is mostly a consequence of high power density. Power density has increased relentlessly over technology generations and will probably continue to increase. One of the reasons is that

Authors' addresses: Pierre Michaud, André Seznec, and Damien Fetis, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France; email: {pmichaud,seznec,dfetis}@irisa.fr. Yiannakis Sazeides and Theofanis Constantinou, Department of Computer Science, University of Cyprus, 75 Kallipoleos Str, 1678 Nicosia, Cyprus; email: {yanos,theofanis}@cs.ucy.ac.cy.

voltage is unlikely to scale proportionally to transistor dimensions, as it used to in recent years [ITRS 2004].

Nonetheless, the reduction of the technology feature size allows several computing cores to be placed on the same chip. This is particularly attractive for server-class processors. Several dualcore processors are presently available on the market and future server class processors will likely have more than two cores (e.g., Sun's eight-core UltraSPARC T1 [Kongetira et al. 2005]).

Power consumption and temperature are important design constraints for existing multicores [Clabes et al. 2004; Poirier et al. 2005; Pham et al. 2005]. Yet, if the product *power density* $\times$ *feature size* increases with future technology generations, temperature will be a stronger constraint than total power consumption [Michaud and Sazeides 2006].

In future temperature-constrained multicores (TCMC), thread migration will be essential for fully exploiting the thermal limit [Skadron et al. 2003; Heo et al. 2003; Powell et al. 2004]. We propose a new thread migration method that maximizes performance under a temperature constraint, while minimizing the number of thread migrations and ensuring fairness between threads. Our method is based on exchanging threads when detecting a pair of hot–cold cores. We confirm previous studies on activity migration and show that thread migration brings important performance gains when performance is constrained by temperature.

Our study brings new insights on thread migration. One of our findings is that thread migration is most effective during the first tens of seconds following a decrease of the number of running threads. We show that the thermal benefit of thread migration depends on the number of threads, their characteristics, and the ambient temperature. We also show that the proposed migration method divides the power consumption among threads in a fair manner.

The paper is organized as follows. In Section 2, we define TCMCs and thermal fairness. Our new migration method is introduced in Section 3. Section 4 presents some experiments that serve to validate our simulation methodology, which we present in Section 5. In Section 6, we show that the thermal benefit of thread migration may be significantly underestimated if one looks only at steady-state behavior and not at transient behavior. Sections 7 and 8 present experimental results quantifying the steady-state and transient performance benefit of thread migration, with a look at migration frequency and fairness. Section 9 presents related work and compares our migration method with HRTM [Powell et al. 2004]. Finally, Section 10 concludes this study.

2. TEMPERATURE-CONSTRAINED MULTICORE

Recent processors implement dynamic thermal management to prevent the absolute temperature from exceeding a certain value. When temperature approaches this value, power and performance are throttled. However, power consumption only impacts the relative temperature, i.e., the temperature rise over the ambient. The absolute temperature is the sum of the ambient temperature T_{amb} and the relative temperature. Hence, for a fixed temperature limit, a lower ambient temperature allows a higher power consumption.

We define a temperature-constrained multicore (TCMC) as a multicore whose power consumption increases when the ambient temperature T_{amb} decreases. We say that a core is *thermally saturated* when temperature on the core is close to the maximum allowed temperature $T_{\max}$. In a TCMC, the maximum temperature $T_{\max}$ is not an emergency temperature that may be occasionally reached. It is a nominal temperature that the chip is likely to reach for a significant fraction of its lifetime. Hence, the value of $T_{\max}$ has an impact on the mean time to failure.

The ambient temperature T_{amb} is the temperature of the inlet air to the heat sink fan [Intel 2004], i.e., inside the computer box. The ambient temperature depends on the box design and is strongly correlated with the room temperature. Hence, the value of T_{amb} is not fixed. For example, the air temperature at different rack inlets in the same air-conditioned room can vary significantly [Moore et al. 2004; Schmidt et al. 2005].

2.1 Thermal Fairness

All the threads running simultaneously on a TCMC contribute to the total power consumption, which is limited by the temperature constraint. Hence, the total power is a shared resource for which threads compete with each other. We define *thermal fairness* as follows:

1. Threads having similar characteristics are given the same share of the total power consumption,
2. A thread X is allotted a power consumption that is not smaller than that allotted if all the other threads were similar to X.

We believe that a practical TCMC should ensure thermal fairness, at least approximately. The operating system (OS) schedules threads according to their respective priorities. In particular, the OS decides which threads should be running at a given time and for how long. However, when threads with similar characteristics are running simultaneously, there is no reason why one should be favored against the other. Hence the first property.

The second property is for preventing a thread from taking up all the power [Hasan et al. 2005]. It guarantees a minimum performance for each thread, independent of other threads characteristics.

2.2 On–Off Thermal Management

A practical TCMC must be able to regulate its power consumption in order to adapt to the ambient temperature and the workload. There exists several possible methods for power regulation, in particular, global clock gating, instruction fetch throttling, and voltage/frequency scaling. Except voltage scaling, these methods involve a linear relation between power and performance.

For this study, we have assumed one of the simplest linear methods, *on–off* thermal management (OOTM), sometimes called *global clock gating*.[1] This method is implemented in the Intel Pentium 4 [Gunther et al. 2001]. When a

[1]But the term is not general enough, as it indicates a particular means. For instance, one may use power gating during off periods.

temperature sensor indicates that $T_{\max}$ is exceeded, OOTM stops the execution on the corresponding core for a fixed time t_{off}, and tries to minimize power consumption during this time. Power density in the region surrounding the sensor is q_{off}. After time t_{off} is elapsed, normal execution resumes for a certain time t_{on}, until the sensor indicates $T_{\max}$ again. During this time, power density in the region surrounding the sensor is q_{on}.

Power density in the vicinity of the temperature sensor is a square wave of period $t_{on} + t_{off}$. This power density oscillation generates a temperature oscillation whose amplitude is approximately [Michaud et al. 2005]:

$$A(\gamma) \approx \frac{q_{on} - q_{off}}{k_1} \times \sqrt{\frac{4\alpha_1 t_{off}}{\pi}} \times \sqrt{\gamma} \tag{1}$$

where k_1 and α_1 are, respectively, the thermal conductivity and diffusivity of silicon (cf. Table II) and the duty cycle $\gamma \in [0, 1]$ is

$$\gamma = \frac{t_{on}}{t_{on} + t_{off}}$$

The duty cycle γ, hence, t_{on} (as t_{off} is fixed), adjusts automatically as a function of q_{on} and ambient conditions. The time-average relative temperature is proportional to the time-average power. Hence, for maximizing the time-average power, we must minimize the amplitude of the temperature oscillation. The value of t_{off} must be chosen small enough for the amplitude of the temperature oscillation to be much smaller than $T_{\max} - T_{amb}$, say less than 1°C. This way, the time-average temperature is close to $T_{\max}$. This requires t_{off} to be much smaller than a millisecond. However, t_{off} must be chosen large enough for the temperature sensor to have enough time to give a new temperature measure (if temperature is still above $T_{\max}$ at the end of the off period, an extra off period is necessary). For example, in the Intel Pentium 4, t_{off} is a few microseconds [Gunther et al. 2001; Intel 2004]. Formula (1) can be used to set t_{off} to an appropriate value. For the experiments in this study, we assume $t_{off} = 10\,\mu$s.

Moreover, we assume that each core has its own OOTM mechanism. In this context, we refine our definition of thermal saturation by considering that a core (or a sensor) is thermally saturated when OOTM triggered recently for that core (or sensor).

3. THREAD MIGRATION METHOD

Activity migration is a general technique that decreases temperature by spreading the electric activity over a larger area [Skadron et al. 2003; Heo et al. 2003; Powell et al. 2004]. At the level of a multicore, activity migration means migrating threads from core to core.

We do not rely on thread migration to keep temperature below $T_{\max}$, but on OOTM. Previous work [Heo et al. 2003; Powell et al. 2004; Shayesteh et al. 2005] proposes migration as an alternative to OOTM or voltage/frequency scaling. However, the extent to which thread migration can decrease temperature is limited, and thread migration cannot replace OOTM. For instance, let us consider a dualcore processor and a single thread running. A simple thread migration method would migrate the thread to the unused core whenever the

active core reaches $T_{\max}$ [Shayesteh et al. 2005], without any other form of thermal management. In this case, if the thread power consumption is very high or in case of very high ambient temperature, the following occurs. Initially, both cores are below $T_{\max}$. As the thread executes and migrates, both cores get hotter. As temperature on both cores approaches $T_{\max}$, the migration frequency gets higher and higher. In theory, the migration frequency becomes infinite and temperature ends up exceeding $T_{\max}$. However, in practice, the migration frequency is limited by the migration performance penalty. In other words, the only reason why temperature could be prevented from exceeding $T_{\max}$ is that the performance penalty from migrations will be sufficiently high to decrease power density. This can be viewed as a form of thermal management, but a very inefficient one, because migrations consume some energy without producing useful work.

With OOTM as our main thermal management mechanism, the goal of thread migration is to increase performance when performance is throttled because of temperature. The total power, hence, the total performance, increases with the spatial-average temperature (cf. Eq. (A4), appendix). Performance is maximum when all cores are thermally saturated, i.e., when the spatial-average temperature of the hottest sensor is approximately $T_{\max}$.

A good thread migration method should avoid situations where a core is thermally saturated while another core is not. A simple way to achieve this would be to migrate threads proactively and periodically, so that all cores "see" the same time-average power density [Heo et al. 2003]. However, migrations induce a performance penalty [Constantinou et al. 2005], so we would like to minimize the number of migrations. The problem is that the optimal migration frequency depends on several parameters. For example, we could imagine a situation where putting the microprocessor in an air-conditioned room at 15°C keeps all cores below $T_{\max}$ and does not require any migrations, while putting the same processor running the same threads in a room at 25°C requires a certain migration frequency. Maybe, if the room temperature is even hotter, all cores become thermally saturated and migrations are useless again. Actually, the optimal migration frequency depends not only on the ambient temperature, but also on packaging characteristics (which may vary during the processor lifetime [Samson et al. 2005]) and threads characteristics.

A migration method that tries to minimize thread migrations should be based on thermal sensor information, as noted in Heo et al. [2003]. We should migrate threads only when there is a potential performance gain. In particular, if there is a thread running on each core and if all cores are thermally saturated and this concerns the same sensor, there is little to expect from migrations. Thread migration is interesting whenever

- there is a thermal sensor that recently triggered OOTM,
- there exists a core on which the same corresponding sensor did not trigger OOTM for a long time.

We propose the following method, which is based on threads exchanges. First, we allow a new migration only after 1 *ms* has elapsed since the last migration.

This is a safeguard for preventing migrations to occur too often. Periodically (we assume every $1\mu s$ in this study), the centralized thermal management unit (TMU) receives new temperature measures from all thermal sensors. We assume that there is a counter associated with each sensor, giving the current value of t_{on} for that sensor, i.e., the time elapsed since the last off period.

A thread is considered *hot* at a sensor if the current value of t_{on} is less than $20 \times t_{off}$, that is, the sensor recently triggered OOTM. A sensor is considered *cold* if its temperature is less than $T_{sat} = T_{\max} - 5\,K$. If the TMU can find a cold sensor such that there exists a thread that is hot at that sensor, the TMU triggers a migration. If there are several possible cold sensors for which there exists a hot thread, the TMU selects the coldest of these cold sensors. Once a cold sensor is selected (hence, a *cold* core), if there exist several hot threads for that sensor, the TMU selects the thread with the smallest t_{on}. This thread is running on a core, which we call the *hot* core. Once the TMU has identified such a pair of cold–hot cores, the migration consists in moving the hot thread to the cold core. If there was a thread previously running on the cold core, this thread is moved to the hot core, that is, the two cores exchange threads. Eventually, the TMU resets the counters measuring t_{on} on both cores. After the migration, no new migration can be performed before 1 ms has elapsed.

The value $T_{sat} = T_{\max} - 5\,K$ for triggering migrations has been chosen not too far from $T_{\max}$, so that we stay close to thermal saturation and to maximum performance, yet not too close to $T_{\max}$, so that cold sensors can be distinguished from hot ones (although we assume perfect sensors in this study).

3.1 Thermal Fairness

The migration method proposed in Section 3 should respect thermal fairness, for the following reasons. Let us consider threads with identical characteristics. When a hot thread is migrated to a cold core, temperature on the new core increases, until temperature at a sensor reaches $T_{\max}$ and then oscillates because of OOTM. Once OOTM triggers, the thermal state continues to evolve (heat sources further from the sensor contribute to sensor temperature after a longer time) and the duty cycle γ decreases progressively to a steady-state value. The longer the thread stays on a core, the smaller the duty cycle. Since threads have identical characteristics, the thread with the smallest duty cycle is the thread that stayed for the longest time without migrating. When we have several possible choices for the hot thread, we select the thread with the smallest t_{on}. This method favors the thread with the smallest duty cycle. That is, the thread that was not migrated for the longest time has a higher probability than the other threads to be the next thread to be selected for migration. The longer the time, the higher the probability. This tends to equalize the threads migration intervals. Hence, identical threads consume approximately the same time-average power.

Now let us consider threads that are not necessarily identical and let us focus on thread X. Thermal fairness means that if we replace the other threads by threads identical to X, the duty cycle of thread X should not increase. If OOTM seldom triggers for thread X, thermal fairness is obviously respected. Thus, let us consider the case when thread X power consumption is throttled. In this

case, temperature at the throttling sensor lies between T_{sat} and $T_{\max}$ on each core. Let q_{sat} be the time-average 2D power density in the region surrounding the sensor. We can distinguish two types of threads: saturating threads, i.e., threads with $q_{on} > q_{sat}$ and for which the sensor triggers OOTM (including thread X), and desaturating threads, i.e., threads with $q_{on} < q_{sat}$ for which the sensor does not trigger OOTM. If we replace all the other threads by threads identical to X, all the threads are now saturating. The time-average power density q_{sat} should stay approximately the same as before, but the duty cycle for thread X should be smaller to compensate for the lack of desaturating threads. Hence, the power consumption allotted to thread X is not smaller than that allotted if all the other threads were similar to X.

4. PRELIMINARY EXPERIMENTS

Experiments in this study are conducted with an abstract model of performance and power consumption that does not take into account microarchitecture details. We justify some of the assumptions of the model with preliminary experiments presented in this section. In Section 4.1, we show that the power dissipated in a processor is roughly a linear function of the number of instructions retired per second (IPS). In Section 4.2, we run trace-driven simulations and study the performance penalty of thread migrations, without considering temperature. In particular, we show that migrating every 2 millions cycles incurs little performance loss, but migrating every 200 thousands cycles may impair performance.

4.1 Relation between IPS and Power Consumption

Our power consumption model is based on the assumption that, for a fixed voltage and frequency, the power consumed by a program execution is strongly correlated with the number of instructions retired per second (IPS) and that it is approximately a linear function of the IPS. Although this is inexact (e.g., branch mispredictions decrease the IPS, but increase the energy consumption), we are looking for a first-order approximation in this study.

To verify the validity of this assumption, we tried to correlate the IPS with the chip temperature by accessing the on-chip thermal diode of a Pentium M.

We use 22 SPEC CPU benchmarks that we run with the reference inputs on an AC-powered Dell Latitude D600 laptop (1400-MHz Pentium M). For each benchmark, we do three runs. In the first and second run, the benchmark is executed to completion. With the first run, we measure the execution time. With the second run, we instrument the benchmark with Pin [Luk et al. 2005] and we obtain the number of instructions executed. The first and second runs give the IPS.

With the third run, we measure temperature. The Pentium M features two on-chip thermal diodes [Rotem et al. 2004], of which only one can be read. This diode is not located at the hottest point on the chip. Nevertheless, the temperature number it provides should be correlated with the total power dissipated on the chip. Before executing each benchmark, we wait for a certain time without executing any command, so that temperature drops and stabilizes to its idle

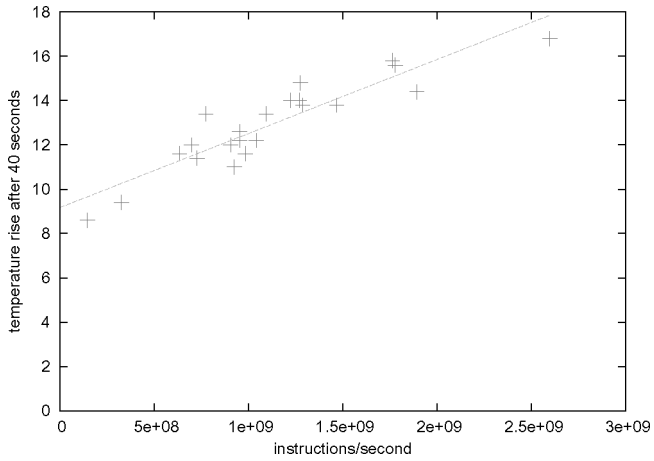

Fig. 1. Compare with Table I. Each point is for one of 22 SPEC CPU 2000 benchmarks executed with the reference inputs on an AC-powered Dell Latitude D600 laptop (1400-MHz Pentium M). The x axis shows the number of X86 instructions retired per second when the benchmark is run to completion. The y axis shows the temperature rise after 40 s of execution, as given by the on-chip thermal diode. Between successive executions, we wait long enough for temperature to return to a stable "idle" value.

value. On the day of the experiment, the idle temperature was approximately 43°C. The idle temperature is the result of the room temperature, the residual power consumption on the chip (static power, nongated clock power), and the power generated by all other electric devices in the laptop. What we measure is the temperature rise when executing a task, i.e., the temperature as given by the ACPI temperature file, minus the idle temperature. The temperature rise should be roughly proportional to the extra power dissipated when we run an application.

We measure temperature after 40 s of execution of each benchmark (more precisely, temperature averaged over a few points around $t = 40$ s). The reason why we do not take the steady-state temperature is that the principle of superposition applies to transient temperature as well [Michaud et al. 2005] (some benchmarks do not run long enough to reach a steady-state). Moreover, after several minutes of execution of a hot application, the fan speed increases, which changes the sink-to-ambient thermal resistance. We do not have this problem with temperature at $t = 40$ s.

The result of this experiment is shown in Figure 1. Each point is for one of the 22 SPEC benchmarks listed in Table I. The x axis shows the IPS, and the y axis the temperature rise (cf. Table I). We have plotted on the graph a least-square linear fit, which is not far from the points. This confirms our assumption that power is strongly correlated with the IPS and that it is approximately a linear function of the IPS, which also confirms previous work [Felter et al. 2005].

It should be noted that there is a nonnull power consumption even when the IPS is very small. This is normal and corresponds to static power and nongated dynamic power (in particular, part of the clock distribution and circuitry).

Table I. SPEC CPU 2000 Benchmarks[a]

Benchmark	GIPS	Temperature Rise (°C) ($t = 40$ s)
181 mcf	0.15	8.6
179 art	0.33	9.4
171 swim	0.63	11.6
300 twolf	0.70	12.0
197 parser	0.73	11.4
183 equake	0.77	13.4
188 ammp	0.91	12.0
177 mesa	0.93	11.0
175 vpr	0.95	12.6
173 applu	0.95	12.2
301 apsi	0.98	11.6
256 bzip2	1.04	12.2
172 mgrid	1.09	13.4
164 gzip	1.22	14.0
252 eon	1.27	14.0
176 gcc	1.27	14.8
254 gap	1.29	13.8
255 vortex	1.47	13.8
253 perlbmk	1.76	15.8
186 crafty	1.78	15.6
168 wupwise	1.89	14.4
200 sixtrack	2.60	16.8

[a] giga instructions per second (GIPS) and temperature rise in degree Celsius (°C) at $t = 40$ s.

4.2 Migration Penalty

Thread migrations incur some performance penalty, mainly because of extra cache misses and branch mispredictions [Constantinou et al. 2005]. We ran some simulations to estimate the performance penalty from migrations when several threads are running concurrently.

4.2.1 *Simulator.* Our simulator is trace-driven, using traces generated with Pin [Luk et al. 2005]. We have one trace per SPEC benchmark listed in Table I. To obtain each trace, we run the application without any instrumentation for several tens of seconds, then we send a signal that triggers instrumentation. Each trace represents 10 millions instructions. We managed to take a sample on which the L2 cache miss rate (in misses per instruction) is not too far from that of the whole application. When reaching the end of a trace during a simulation, we wrap around the trace.

We simulate four cores, each with dedicated L1 and L2 caches. Cores communicate through a unidirectional pipelined ring whose clock cycle is four times the CPU clock cycle. The next level after the L2 cache is the off-chip DRAM. In order to stress the performance penalty because of migrations, we do not consider any on-chip shared L3 cache.

The simulator does not model all the microarchitecture details. Basically, the simulator consists of a set of finite-size queues. We have abstracted the execution core with a queue representing a reorder buffer of 128 instructions. The

minimum latency in the reorder buffer is 10 cycles, which models a minimum branch misprediction penalty. In each cycle, one X86 instruction can be retired from the instruction window and two instructions can be fetched into it. We simulate a 12-KB YAGS [Eden and Mudge 1998] with a 25-branch global history. We assume a mispredicted branch is resolved when the instruction window is completely drained (i.e., the actual branch misprediction penalty can be much higher than 10 cycles).

Though we do not take into account data dependencies in our simplified model, we simulate the memory hierarchy. An instruction cannot be retired from the instruction window before all older loads and stores have been executed. We consider a unique cache block size of 64 bytes. The L1 instruction and data caches are both 32 KB, four-way set-associative. The per-core L2 cache is 512 KB, eight-way set-associative. The L1 instruction cache can deliver one cache block per cycle. The L1 data cache and the L2 cache both use a write-back, write-allocate policy. The L1 data cache is nonblocking: when there is a load/store miss in the L1 data cache, subsequent loads/stores can execute and possibly generate miss requests, which are fully pipelined. At most, a single load/store can be executed per cycle.

We do not maintain the inclusion between L1 and L2 caches. Before being sent to memory, a request that misses both the L1 and L2 is sent on the ring and we search for the missing block in other cores caches and write-back queues. We assume that cache tags are replicated so that snooping does not impact the performance of the inspected core when the requested block is missing. We allow cache-to-cache accesses not only for dirty data blocks, as required, but also for clean data and instructions blocks. In case the request can be satisfied by another core, the request is removed from the ring and steals some cache bandwidth, if necessary, to get the block. The request is intercepted by the first core encountered on the ring that can satisfy the request. The ring can deliver 16 bytes per core and per CPU cycle.

If the request goes round the ring and returns to the core that emitted it without having been intercepted, the request is sent to DRAM. The bus bandwidth to DRAM is four bytes per CPU cycle. There are separate queues for requests from different cores. The bus arbiter prioritizes requests to give bus access to cores that have been served least recently. Once a request is sent on the bus, there is a latency of 200 cycles for getting the requested block. Bus request queues are made large enough so that the bottleneck is not queuing, but bus bandwidth (i.e., the number of queue entries is greater than $200 \times 4/64 = 12.5$). The minimum latency for a L2 miss served by DRAM is $200 + 4 \times 4 = 216$ cycles, where the 16 extra cycles is the time for a request to go round the ring.

Cache coherency is maintained by invalidating matching blocks on remote cores when writing in the L1 data cache. When there are several copies of a cache block on different cores, we assume that, at most, a single copy can be marked dirty. When forwarding a dirty block on a L2-to-L2 miss, the copy is considered clean, and the original block remains dirty.

We implemented a L2-prefetcher that is alike the one implemented in the IBM POWER4 [Tendler et al. 2002]. When arbitrating between requests for a resource (caches, ring, bus), demand requests have priority over prefetch requests.

4.2.2 *Simulations.* The first set of simulations examines the performance when running four independent threads on fixed cores, that is, thread #0 on core #0, thread #1 on core #1 and so on. We simulate 100 millions CPU cycles. Our performance metric is the IPC of thread #0, which we denote IPC_0. This represents the total number of X86 instructions from thread #0 that have been retired, divided by 100 millions. Each thread is given a different thread ID (TID), which is used to distinguish address spaces. The TID is part of the cache tags and is also used in the branch predictor.

For each benchmark listed in Table I, we run 30 simulations, where the benchmark studied is thread #0 and the three other threads are taken (pseudo-) randomly out of the set of 22 benchmarks. Each simulation represents 100 millions CPU cycles. The pseudorandom list of threads is always generated with the same seed, so that it is the exact same list for each set of 30 simulations. At the end of the simulation, we compute the arithmetic mean of IPC_0 over the 30 simulations. The resulting average IPC_0 is shown as the left bar for each benchmark on Figure 2 (note that the IPC is artificially limited to 1 by our simulator).

We next evaluate periodic migrations. Periodically, we move thread #0 from core $m \in [0, 3]$ to core $n = (m + 1) \bmod 4$. The thread that was running on core n is moved to core m. That is, we exchange threads. Once a migration is triggered, we wait until both threads have their instruction window completely drained before actually switching threads. Thus, there is an immediate penalty which is approximately that of a branch misprediction. Then there is a penalty for switching program states. This penalty, which we did not model, should be negligible compared with the migration interval [Constantinou et al. 2005], especially if saving and restoring register values is done by the firmware.

The second bar for each benchmark on Figure 2 shows IPC_0 averaged over the 30 simulations when migrating every 2 millions CPU cycles. We observe that migrating thread #0 every 2 millions cycles has little impact on IPC_0. For some benchmarks, like 175, 179, and 300, we even observe a nonnegligible performance increase, which is a result of DRAM accesses that have been turned into cache-to-cache misses (that have a smaller latency and a higher bandwidth). If we migrate 10 times more frequently, i.e., every 200 thousands instead of 2 millions cycles, we observe only a slight performance loss on most benchmarks (third bar). Yet, it is interesting to note that the performance of benchmark 179 is significantly higher when migrating. The main performance bottleneck for benchmark 179 is L2 cache size and memory bandwidth. Migrations help alleviating this bottleneck by allowing the thread corresponding to benchmark 179 to distribute its data over several L2 caches. Nevertheless, exploiting this phenomenon is out of the scope of this study.

The three left bars for each benchmark on Figure 2 are for threads #1, #2, and #3 taken randomly in the list of 22 benchmarks. However, thread #0 is penalized by certain threads more than by others. Among our list of 22 benchmarks, the less "friendly" one is benchmark 179, which has a large data working set that is swept very quickly.

We repeated the same experiments as previously but, instead of running 30 simulations for each benchmark, we run a single one with threads #1,#2,

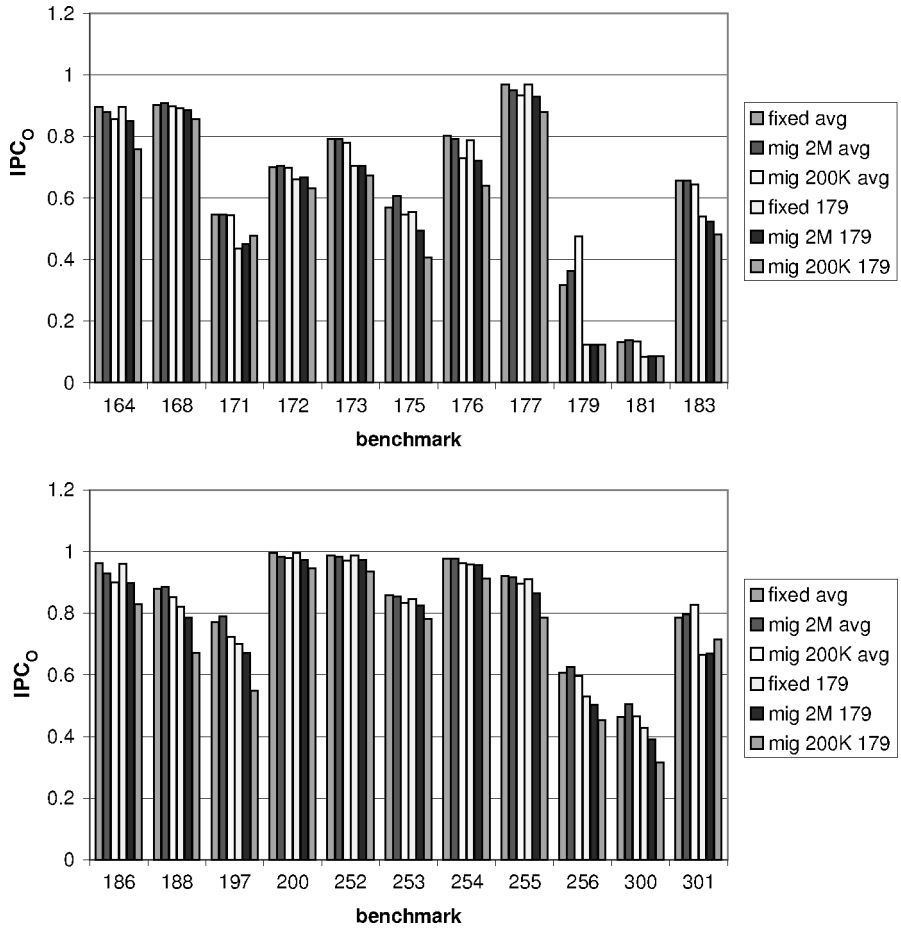

Fig. 2. Simulated IPC for each benchmark.

and #3 being benchmark 179 (fourth, fifth, and sixth bars). We recall that each thread has a different TID. This situation represents a worst case, as thread #0 has most of its data and instructions evicted from remote caches by the other threads. Moreover, when accessing DRAM, it has to compete with the other threads for bus bandwidth. As can be observed, migrating every 200 thousands cycles incur a significant performance loss on several benchmarks (164, 175, 176, 177, 183, 186, 188, 197, 255, 256, 300).

The migration method introduced in Section 3 prevents migrations to occur until 1 ms elapsed since the last migration. With a 5-GHz CPU clock, 1 ms represents 5 millions cycles. From the results of Figure 2 (second bar versus first one, fifth bar versus fourth one), we expect that, when performance is limited by temperature, the thermal benefit of migrations outweighs their performance cost. The discussion in this section is for cores having private L2 caches and no L3 cache. If there is an on-chip shared L3 cache, the migration penalties will be even smaller than those exhibited on Figure 2. In the

remainder of this study, we neglect the migration penalty and focus on thermal aspects.

5. MODELING METHODOLOGY

We model the instruction execution throughput E_{on} of a thread in the *on* state as

$$E_{on} = \frac{1}{M + C/f} \tag{2}$$

where E_{on} is in instructions per second, f is the clock frequency in hertz, C is in cycles per instruction, and M is the average time, in seconds per instruction, spent waiting for off-chip accesses. When the duty cycle γ is less than 1, the actual performance is $E = \gamma E_{on}$.

5.1 Power and Temperature Model

We model the power density q_{on} in the core as

$$q_{on} = q_s + \beta_u \times E_{on} + \beta_c \times f \tag{3}$$

where q_s is the static power density, $\beta_u \times E_{on}$ is the dynamic power density corresponding to the energy spent doing useful work, and $\beta_c \times f$ is the contribution from the clock circuitry. Parameters q_s, β_u, and β_c depend on voltage. In this study we consider a constant voltage, hence, constant parameters. The contribution $\beta_c \times f$ from the clock circuitry is important on current processors. For instance, on the Intel Itanium Montecito, about 25% of the total power consumption comes from the clock circuitry [Naffziger et al. 2005]. Clock-gating techniques can mitigate this problem, but cannot solve it completely [Jacobson et al. 2005].

We assume that when a core is not executing instructions, power density is null, i.e., $q_{off} = 0$. Indeed, it is possible to bring power density to a low value, provided the off period is long enough. Clock gating removes dynamic power consumption and power gating with high V_t sleep transistors can remove most static power consumption, provided it is applied for a sufficiently long time, e.g., several microseconds [Tschanz et al. 2003]. The power dissipation remaining in the core is essentially the power necessary to preserve data in the various tables (branch predictor, TLB, level-1 cache, register file, ...). In the remaining, we neglect this power consumption [Nii et al. 1998; Kim et al. 2004]. In any case this assumption does not change our qualitative conclusions.

All temperature numbers in this study were obtained with ATMI [ATMI; Michaud et al. 2005]. We used the ATMI parameters listed in Table II. We model the four-core chip as depicted on Figure 3. Each core is modeled as a 3-mm side square dissipating a uniform power density. The rest of the chip is assumed to dissipate a power that is negligible compared with the total power dissipated by cores.[2] We consider a single thermal sensor per core, located at the center of each core. For the maximum allowed temperature $T_{\max}$, we assume $T_{\max} = 85^{\circ}$C

[2]Even if not negligible, this power consumption brings roughly the same temperature contribution on all cores, via an increase of the heat-sink temperature.

Table II. Parameter Values Used in this Study

Parameter	Value	Unit	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	4×10^4	W/m^2K	Interface material thermal conductance
h_2	500	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

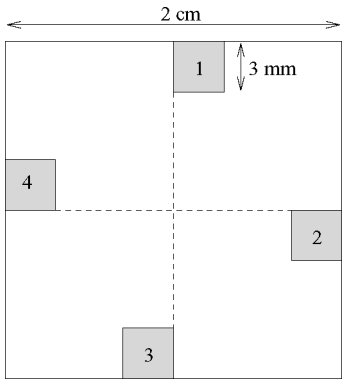

Fig. 3. Four-core chip model. Cores are modeled as 3-mm side squares dissipating a uniform power density. The rest of the chip is assumed to dissipate a negligible power.

[ITRS 2004]. The clock frequency and parameters for power density are given in Table III. These parameter values were chosen such that an IPC of one instruction per cycle ($E_{on} = 1 \times f = 5$ GIPS) generates a power density $q_{on} = 2$ W/mm^2, 25% of which is static power and another 25% is from the clock circuitry [Naffziger et al. 2005].

5.2 Artificial Threads

In following experiments, we use artificial threads whose characteristics are listed in Table IV. We distinguish hot, warm, and cold threads. We generate instructions artificially according to specified values C and M (formula 2). Power density q_{on} is obtained from Formula (3).

It should be noted that a given application may be warm or cold, depending on the number of threads that are running concurrently and depending on their characteristics. For instance, if bus contention limits the IPS, the more memory-bound threads are running concurrently, the cooler the threads. However, our study is not concerned with this question, but with understanding the thermal benefit of thread migrations for a given set of threads running concurrently.

Table III. Clock Frequency, Power Density Parameters, and Maximum Temperature

f	5.10^9 Hz
q_s	0.5 W/mm^2
$\beta_c \times f$	0.5 W/mm^2
β_u	2.10^{-4} J/m^2
$T_{\max}$	85°C

Table IV. Different Types of Artificial Threads

Type	C	$M \times f$	E_{on} (GIPS)	q_{on} (W/mm^2)
Hot	1/3	0	15	4
Warm	1	0	5	2
Cold	1	10	0.45	1.1

5.3 Initialization and Simulation Time

After a long period of low activity, the temperature of the heat sink is close to local ambient. The performance of a TCMC will be high in the first seconds following such idle period. If a high activity is maintained for several minutes, the performance decreases progressively, until the heat sink temperature reaches a steady state. When we study steady-state performance, TCMC simulations must be put in a meaningful initial state. There are several possibilities. The most rigorous way would be to study the performance of the system over a period of time long enough to reach a steady state. If one does not know workload characteristics over such a long period (or if simulations are too slow), it is possible to extrapolate the workload behavior, i.e., to assume that workload characteristics will not change [Skadron et al. 2003; Srinivasan and Adve 2005].

In this study, we assume that at $t = 0$ the four cores are thermally saturated. In a TCMC, assuming that cores are thermally saturated corresponds both to a worst and a likely case. The power density that we apply on each core during the initialization phase is

$$q_{sat} = \frac{T_{\max} - T_{amb}}{R_{ja} \times N \times A_{core}}$$

where $A_{core} = 9\,mm^2$ is the core area and R_{ja} is defined in the appendix. We obtain $q_{sat} = 1.17, 1.46, 1.75$ W/mm^2 for $T_{amb} = 45, 35, 25$°C, respectively.

5.4 Performance

Performance numbers are given either in instructions per second or, when simulating a fixed time, in raw number of instructions executed. As mentioned previously, we do not model any migration penalty.

6. IMPORTANCE OF STUDYING TRANSIENT BEHAVIORS

Figure 4 shows the performance for a single warm thread running on a fixed core, as a function of time, at $T_{amb} = 35$°C. For $t \leq 0$, all cores are thermally saturated. For instance, the processor could have been running four threads

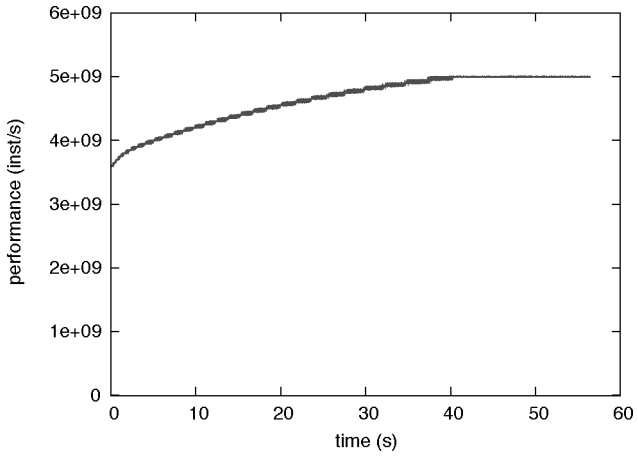

Fig. 4. Single warm thread running on a fixed core. All cores are thermally saturated at $t = 0$. The curve shows performance in instructions per second as a function of time, at $T_{amb} = 35°C$.

for $t < 0$ (for a long time) and then, at $t = 0$, three threads finish and a single thread remains.

For $t > 0$, the temperature of the three inactive cores falls slowly. As the three inactive cores dissipate no power, the active core should be able to dissipate more power. Thus, the remaining thread should run faster after $t = 0$. Actually, it takes a long time until the remaining thread can exploit the benefit of three cores going inactive. Figure 4 shows that the performance of the thread increases with time, but slowly. Initially, the performance is $E = \gamma E_{on} \approx 3.6$ GIPS. This corresponds to the steady-state performance we would obtain if we kept applying power density q_{sat} on cores that are now inactive. The long-term, maximum performance is $E = E_{on} = 5$ GIPS, but it takes about 40 s to reach this performance. For $t > 40$ s, the active core is no longer thermally saturated.

If we enable thread migrations, this phenomenon disappears, i.e., the performance of the remaining thread reaches its maximum value almost immediately. Indeed, if $q_{on} \geq 4q_{sat}$, we are just prolonging the steady state assumed for $t < 0$, i.e., thanks to migrations, the time-average power density is q_{sat} on each core, and cores remain thermally saturated for $t > 0$. Yet, the remaining thread runs four times faster after $t = 0$. On the other hand, if $q_{on} < 4q_{sat}$ (which is the case on this example), the time-average power density, $q_{on}/4$, is smaller than q_{sat}. Hence, temperature drops below $T_{\max}$ and OOTM no longer throttles performance, i.e., the performance of the remaining thread reaches its maximum level very quickly, after a few migrations.

This experiment shows that the full benefit of activity migration cannot be exposed if one looks only at steady state. The benefit of thread migrations may be limited in time. Yet this transient benefit may last as long as several tens of seconds. The transient benefit of thread migration will manifest whenever some extra power can be allotted to the running threads, in particular, when there is a decrease of the number of running threads. Thread migration permits allotting this extra power very quickly.

7. IDENTICAL THREADS

In this section, we consider threads with identical (and fixed) characteristics C and M.

7.1 Steady State

When core 1 is thermally saturated, we have (Eq. A1, appendix)

$$T_{\max} - T_{amb} = A_{core} \times \sum_{j=1}^{4} q_j w_{1j}$$

As $w_{13} \approx w_{12} = w_{14}$, the steady-state duty cycle γ is approximately the same on all active cores. If n threads are running on fixed cores 1 to n, the steady-state time-average power density on active cores is $q_j = \gamma q_{on}$. We have

$$\frac{E}{E_{on}} = \gamma = \frac{T_{\max} - T_{amb}}{A_{core} \times q_{on} \times \sum_{j=1}^{n} w_{1j}}$$

which gives the steady-state relative performance as a function of T_{amb}. As expected, when T_{amb} decreases, performance increases until $\gamma = 1$.

On the other hand, when migrations are enabled, all four cores are active and the time-average power density is $q_j = \gamma q_{on} \times n/4$. We have

$$\frac{E}{E_{on}} = \gamma = \frac{T_{\max} - T_{amb}}{A_{core} \times q_{on} \times R_{ja} \times n}$$

Figure 5 shows the steady-state performance gain of enabling versus disabling migrations as a function of T_{amb} when fewer threads than cores are running. For hot threads, enabling migrations is always beneficial in the range of realistic ambient temperatures, and the performance gain is important, e.g., more than 50% for two threads. For warm threads, the benefit of migrations is less pronounced. In the range $T_{amb} \in [25, 45]^\circ$C, migrating three threads is always beneficial, with up to 20% extra performance from migrations. With two warm threads, enabling migrations is interesting only for $T_{amb} > 30^\circ$C, but the potential performance benefit is relatively high. For a single warm thread, enabling migrations is interesting only for ambient temperatures above 40°C.

We recall that the results on Figure 5 concern the steady-state performance and we have seen that it may take several tens of seconds to reach this performance (cf. Figure 4).

7.2 Transient Behavior

We now consider the transient behavior over a simulated time $t_{sim} = 0.1$ s, and for warm threads. Figure 6 shows the total number of instructions executed by the multicore during $t_{sim} = 0.1$ s as a function of the number of threads, with and without thread migration, for $T_{amb} = 45, 35, 25^\circ$C. Unlike what was observed for the steady-state performance, enabling migrations when a single warm thread is running is always beneficial for transient performance in the considered range of ambient temperature. This fact generalizes to multiple threads, as long as there are less threads than cores. When migrations are

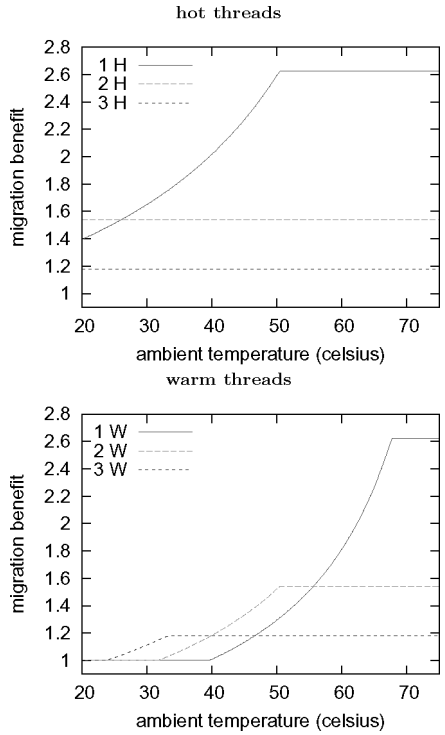

Fig. 5. Steady-state relative performance of enabling versus disabling migrations as a function of T_{amb} when fewer threads than cores are running. The upper graph is for hot threads, while the lower graph is for warm threads. A value of 1 indicates no performance benefit.

disabled, transient performance is proportional to the number of threads. Once again, this is unlike the steady-state performance. On the other hand, with migrations enabled, transient performance equals steady-state performance. This experiment confirms what was stated in Section 6: thread migration is most beneficial in the period following a decrease of the number of running threads.

It should be noted that at $T_{amb} = 45^{\circ}$C and with migrations enabled, adding a third thread increases the total throughput only slightly. This is because cores are already close to thermal saturation with only two threads (the time-average power density on each core is 1 W/mm^2, which is close to q_{sat}). Actually, the total throughput with three threads and migrations enabled is almost as high as with four threads and no migration.

In other words, when temperature limits performance, thread migrations permit maximizing throughput with fewer threads. This fact may be exploited by the OS when threads have different priorities [Michaud and Sazeides 2006].

7.2.1 *Migration Frequency.* Table V shows the number of migrations that have occurred for each thread during $t_{sim} = 0.1$ s. The migration frequency depends on the number of threads and on the ambient temperature, but in a nontrivial way. For instance, for $T_{amb} = 45^{\circ}$C, the migration frequency is

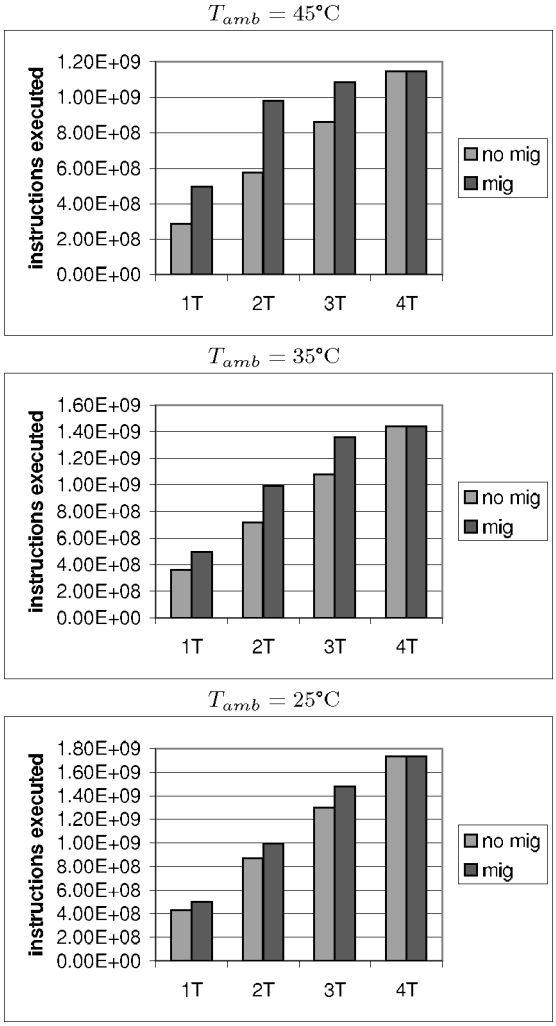

Fig. 6. Total number of instructions executed, as a function of the number of warm threads, with and without thread migrations enabled, for $T_{amb} = 45, 35, 25^{\circ}$C and $t_{sim} = 0.1\,s$.

highest when two threads are running (the average time between consecutive migrations is $t_{sim}/41 \approx 2.4$ ms for each thread). This corresponds to a situation where enabling migrations brings a large performance potential (Figure 6, $T_{amb} = 45^{\circ}$C, "2T"). On the other hand, with a third thread ("3T"), cores are thermally saturated. The relative performance gain from enabling migrations is still significant, but not as pronounced as with two threads. We see in Table V that this corresponds to a smaller number of migrations. Our migration method reduces automatically the migration frequency when there is less benefit to expect from migrating.

Figure 6 shows only the aggregate performance. However, we verified that when several threads are running with migrations enabled, thermal fairness is

Table V. Number of Per-Thread Migrations for the Experiments Reported on Figure 6

# Threads	Number of Migrations		
	Thread 1	Thread 2	Thread 3
$T_{amb} = 45^{\circ}$C			
1	14		
2	41	41	
3	19	10	12
$T_{amb} = 35^{\circ}$C			
1	9		
2	11	12	
3	20	22	19
$T_{amb} = 25^{\circ}$C			
1	5		
2	5	6	
3	27	15	22

respected, i.e., threads have approximately the same performance (and power consumption). Yet, the per-thread migration frequency is not necessarily the same for all threads, e.g., at $T_{amb} = 45^{\circ}$C with three threads. Nevertheless, this is only an initial effect. We verified that the per-thread migration frequencies tend to equalize as t_{sim} increases. Statistically, our thread migration method treats similar threads identically.

8. MIX OF THREADS WITH DIFFERENT CHARACTERISTICS

We use strings to represent thread mixes. For example, *HWC* means that only three threads are running: thread one is a hot thread, thread two is warm, and thread three is cold. We study the three-thread mix *HWC* and the 4-thread mixes *HWCC*, *HWCW* and *HWCH*.

Figure 7 shows the transient ($t_{sim} = 0.1$ s) aggregate throughput, with and without thread migrations. We observe that when there is a thread running on each core, enabling migrations does not bring any benefit when $T_{amb} = 45^{\circ}$C. Actually, no migration occurs here, because even without migrations, cores are thermally saturated or very close to thermal saturation (for cold threads, q_{on} is close to q_{sat}). However, for $T_{amb} = 35^{\circ}$C and 25°C, q_{on} for cold threads is smaller than q_{sat}, and the potential benefit from migrations is significant. This benefit is more pronounced when there are two cold threads.

Figure 8 shows the performance for each thread at $T_{amb} = 25^{\circ}$C. We see that most of the benefit from enabling migrations goes to the hot thread. Figure 9 shows the mapping of threads on cores for the *HWC* mix as a function of time, at $T_{amb} = 25^{\circ}$C. It shows that most migrations (about 90% of migrations) involve the hot thread and the inactive core. On this example, the cold thread migrates from time to time, but less often than the hot thread. However, this is not the case for the four-thread mixes. For instance, for *HWCH*, the thread that migrates most often is the cold thread, because this is the only thread whose q_{on} is less than q_{sat}. Nevertheless, on this example, the migration frequency of the cold thread is still low, with more than 6 ms between consecutive migrations, on average. By comparing *HWC* with the four-thread mixes, we can see the effect

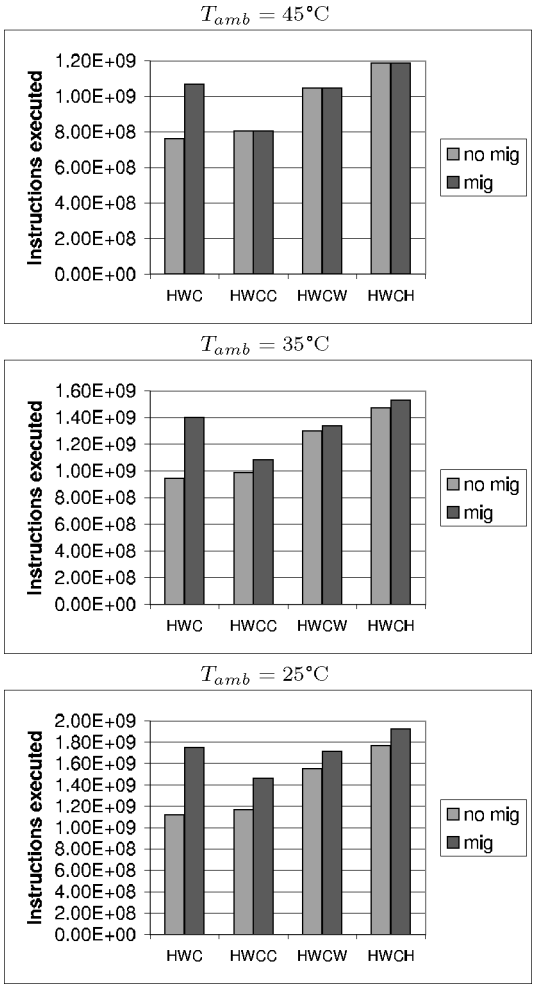

Fig. 7. Transient ($t_{sim} = 0.1\,s$) aggregate throughput for various thread mixes, with and without thread migrations enabled.

of adding a fourth thread. In all cases, the power consumption allotted to the fourth thread is mostly taken from the hot thread. This is an illustration of thermal fairness.

We verified that thermal fairness was approximately respected: similar threads have approximately the same performance, and the performance of a thread X is not smaller than that obtained when all threads are similar to X.

9. RELATED WORK

The problem of process scheduling under a temperature constraint has been previously studied for a single-core processor [Rohou and Smith 1999]: when temperature exceeds the maximum allowed temperature, the OS tries to identify the hot processes responsible for the high temperature, which are usually

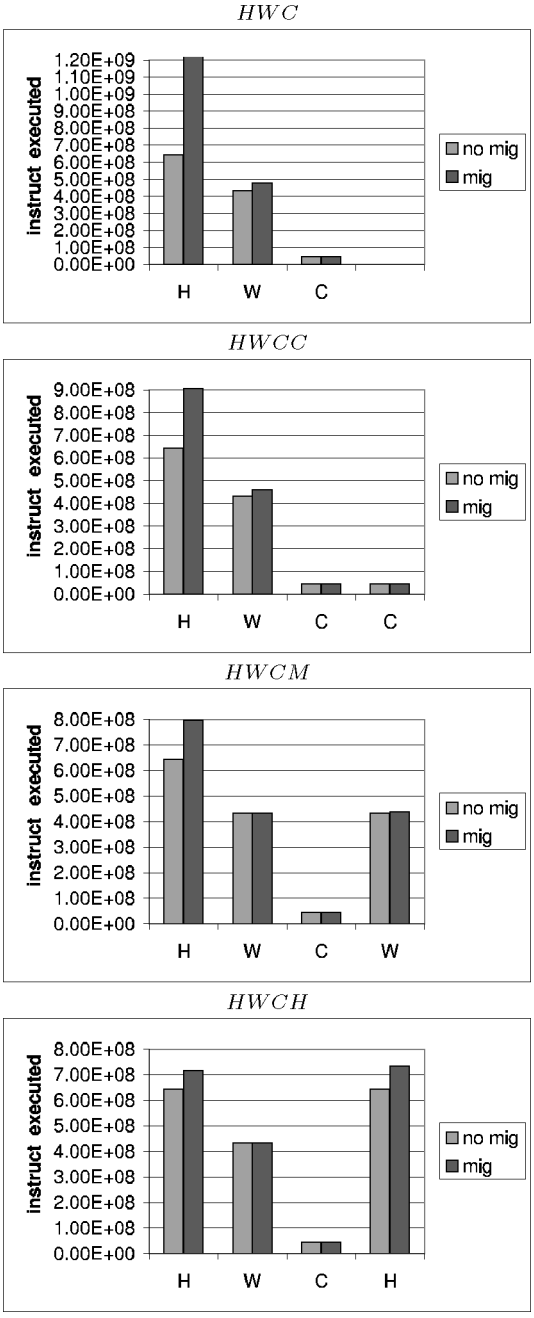

Fig. 8. Transient throughput for each thread, at $T_{amb} = 25°\text{C}$.

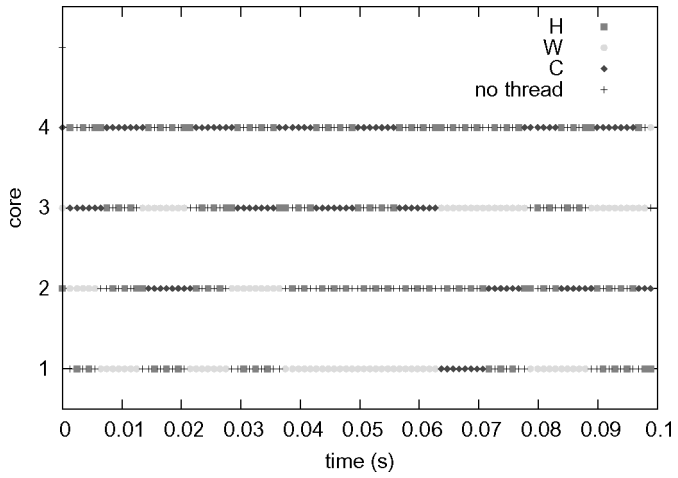

Fig. 9. Mapping of threads on cores for the *HWC* mix, at $T_{amb} = 25^{\circ}$C.

CPU-bound, and penalizes them. Interactive processes and I/O bound processes are not penalized.

The general idea of activity migration for alleviating the temperature constraint has been studied in Lim et al. [2002], Skadron et al. [2003], and Heo et al. [2003]. Thread migration as a special case of activity migration has been studied in Powell et al. [2004] and Shayesteh et al. [2005]. In Shayesteh et al. [2005], the case of a single thread running on a dualcore is considered. The thread is migrated to the unused core whenever temperature reaches the temperature limit. However, there is no mechanism for limiting the migration frequency, which accelerates as time increases, as we explained in Section 3.

The question of how to distribute threads in a multicore under a temperature constraint has been studied in Powell et al. [2004] for dual-threaded cores with idle contexts. A migration method, called HRTM, is introduced. When a core reaches the temperature limit, threads on that core are moved to other cores. The hot core is allowed to cool down for a fixed duration before accepting threads again. This method permits limiting the migration frequency and is similar to ours in that respect. However, HRTM uses migration as a replacement for OOTM (called *stop–go* in Powell et al. [2004]). Powell et al. [2004] does not mention what action is taken whenever several cores are cooling down simultaneously and the number of contexts remaining exceeds the number of threads.

We did a simple experiment to illustrate the difference between our migration method and HRTM. Although Powell et al. [2004] considered SMT cores, HRTM can be easily extended to the case of single-threaded cores with less threads than cores. When temperature reaches $T_{\max}$ on a core, this core is turned off for a fixed duration and we move the thread to an unused core. If the destination core is also in a cooling period, we migrate the thread anyway, and the thread waits there and resumes its execution after the cooling period is over.

We consider three hot threads running on four cores, and we run experiments similar to those in Section 7.2, assuming $T_{amb} = 45^{\circ}$C. Table VI shows the total

Table VI. Three Hot Threads at $T_{amb} = 45^{\circ}$C for $t_{sim} = 0.1$ s

	Total Throughput (# instructions)	# Migrations (total three threads)
OOTM only	1.3×10^9	0
OOTM + migrations	1.6×10^9	43
HRTM cooling = 1 ms	1.4×10^9	308
HRTM cooling = 8 ms	9.9×10^8	44
HRTM cooling = 100 μs	1.6×10^9	2911

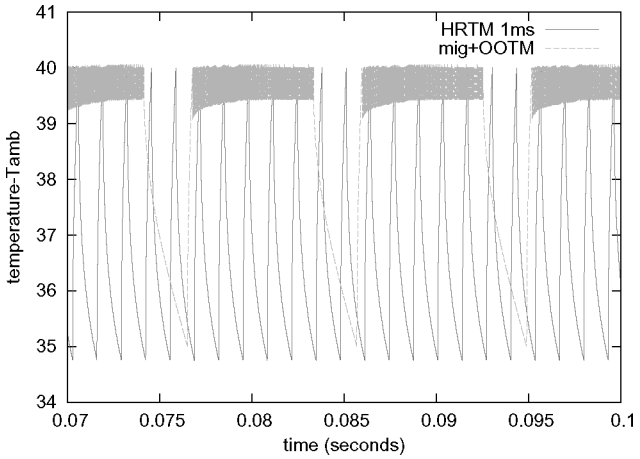

Fig. 10. Three hot threads at $T_{amb} = 45^{\circ}$C. Temperature (relative to ambient) on a core as a function of time. Comparison between HRTM and our method.

throughput for OOTM only (no migrations), for OOTM and migrations (our method), and for HRTM with different cooling durations.

When the cooling duration is 1 ms, HRTM increases performance over the no-migration case (about 8% more throughput). However, our method yields about 23% more throughput compared with the no-migration case. Moreover, our method requires much less migrations, as shown in the last column of Table VI. A possible way to decrease the number of migrations with HRTM is to increase the cooling duration. With a cooling duration of 8 ms, HRTM generates approximately the same number of migrations as our method. However, in this case, the performance is significantly degraded. The last row of Table VI is for HRTM with a cooling duration of 100 μs. In this case, HRTM provides the same throughput as our method, but requires a much larger number of migrations, whose performance penalty may be significant, as we have shown in Figure 2.

The reason why our method is more efficient than HRTM is placed into evidence in Figure 10, which shows a snapshot of temperature on a core as a function of time. With HRTM, cores are periodically on and off, which generates a temperature oscillation, as with OOTM. However, HRTM uses a much longer t_{off} than OOTM, which makes the amplitude of the temperature oscillation larger, and the time-average temperature significantly smaller than $T_{\max}$. That is, the thermal limit is underexploited.

10. CONCLUSION

Previous studies have shown that thread migration is an efficient technique for increasing performance under a temperature constraint [Heo et al. 2003; Powell et al. 2004], which our study confirms. Our simulations indicate that the performance loss induced by migrations on a four-core processor may be significant if a thread does not stay longer than a few hundreds of thousands of cycles on a core before migrating to a new core. Thus, a safe migration method should include a safeguard for limiting the migration frequency.

We have proposed a new thread migration method for TCMCs: we exchange threads whenever the simultaneous occurrence of a cold and a hot core is detected. Under a thermal constraint, this method permits maintaining cores close to thermal saturation, hence to maximum performance, while minimizing the number of thread migrations.

We have shown that the benefit of thread migration is not constant in time. Thread migration is most beneficial in the first tens of seconds following a decrease of the number of running threads.

This study was not concerned with determining how many threads and which threads should be running simultaneously, which is an OS issue. Thread migration is able to increase throughput when there is a thread running on each core, provided threads have different thermal characteristics. However, thread migration brings all its potential when there are fewer running threads than cores. We see at least two reasons why the OS may run fewer threads than cores at a given time:

1. There are not enough runnable threads at that time.
2. Threads have different priorities and the performance of a given thread depends on how many threads are running concurrently. For example, it there are four cores and only four runnable threads with different priorities, running a thread on each core does not give to each thread a performance reflecting its priority. We have shown in this study that when the total throughput is limited by temperature, it is sometimes possible to run fewer threads with little loss of total throughput, thanks to thread migration. This gives the OS a degree of freedom for adjusting the fraction of CPU time given to each thread so that its performance reflects its priority [Michaud and Sazeides 2006].

In both cases, ensuring fairness between threads running concurrently is important. Our thread migration method respects thermal fairness as we defined it. The OS can rely on this predictable behavior to take proper scheduling decisions.

APPENDIX

This appendix provides a simple model for reasoning about steady-state temperature in multicores.

Let us consider a processor with N identical cores, numbered from 1 to N. Let P_i be the time-average power dissipated by core i. For simplicity, we neglect the power dissipated outside cores, e.g., shared caches and buses. From the

principle of superposition, the steady-state time-average relative temperature w_i in core i is

$$w_i = \sum_{j=1}^{N} P_j \times w_{ij} \tag{A1}$$

where $w_{ij} > 0$ is the steady-state relative temperature in core i per watt generated in core j. It should be noted that $w_{ij} \approx w_{ji}$, i.e., matrix (w_{ij}) is approximately symmetric. This stems from the principle of reciprocity [Michaud et al. 2005].

Let us apply the same power $P_i = P/N$ on each core, with P the total power. Relation (A1) yields

$$w_i = \frac{P}{N} \times \sum_{j=1}^{N} w_{ij}$$

We say that cores are *equivalent* if w_{ii} and the sum

$$R_{ja} = \frac{1}{N} \sum_{j=1}^{N} w_{ij} = \frac{1}{N} \sum_{j=1}^{N} w_{ji}$$

do not depend on i. In this case

$$w_i = P \times R_{ja} \tag{A2}$$

Quantity R_{ja}, measured in kelvins/watt, is commonly called *junction-to-ambient thermal resistance*. The equivalence of cores implies that when the same power is applied on each core, all cores have the same temperature. It also implies that all the rows of the symmetric matrix (w_{ij}) are obtained by a permutation on the first row. Hence, the N values w_{1j} are sufficient to obtain the temperature on each core. For the four-core chip depicted on Figure 3, and for the parameter values given in Table II, we have

$$\begin{aligned} w_{11} &\approx 2.52\,K/W, \\ w_{12} &\approx 0.44\,K/W, \\ w_{13} &\approx 0.42\,K/W, \\ w_{14} &= w_{12}, \\ R_{ja} &= \frac{1}{4} \sum_{j} w_{1j} \approx 0.95\,K/W \end{aligned}$$

Relation (A2) can be generalized to cores with different power consumptions, provided we consider the spatial-average temperature defined as

$$\overline{w} = \frac{1}{N} \sum_{i=1}^{N} w_i \tag{A3}$$

From (Eq. A1), we have

$$\overline{w} = \sum_{j=1}^{N} P_j \frac{1}{N} \sum_{i=1}^{N} w_{ij} = P \times R_{ja} \tag{A4}$$

where $P = \sum_{j=1}^{N} P_j$ is the total power.

REFERENCES

ATMI. http://www.irisa.fr/caps/projects/ATMI/.

CLABES, J., FRIEDRICH, J., SWEET, M., DILULLO, J., CHU, S., PLASS, D., DAWSON, J., MUENCH, P., POWELL, L., FLOYD, M., SINHAROY, B., LEE, M., GOULET, M., WAGONER, J., SCHWARTZ, N., RUNYON, S., GORMAN, G., RESTLE, P., KALLA, R., MCGILL, J., AND DODSON, S. 2004. Design and implementation of the POWER5 microprocessor. In *Proceedings of the 41st Design Automation Conference*.

CONSTANTINOU, T., SAZEIDES, Y., MICHAUD, P., FETIS, D., AND SEZNEC, A. 2005. Performance implications of single thread migration on a chip multi core. *ACM SIGARCH Computer Architecture News 33,* 4 (Nov.), 80–91.

EDEN, A. AND MUDGE, T. 1998. The YAGS branch prediction scheme. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*.

FELTER, W., RAJAMANI, K., KELLER, T., AND RUSU, C. 2005. A performance-conserving approach for reducing peak power consumption in server systems. In *Proceedings of the 19th ACM International Conference on Supercomputing*.

GUNTHER, S., BINNS, F., CARMEAN, D., AND HALL, J. 2001. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal 5,* 1 (Feb.).

HASAN, J., JALOTE, A., VIJAYKUMAR, T., AND BRODLEY, C. 2005. Heat stroke : power-density-based denial of service in SMT. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*.

HEO, S., BARR, K., AND ASANOVIĆ, K. 2003. Reducing power density through activity migration. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

INTEL. 2004. Intel Pentium 4 processor on 90nm process thermal and mechanical design guidelines. Document 300564.

ITRS. 2004. International technology roadmap for semiconductors. http://www.itrs.net.

JACOBSON, H., BOSE, P., HU, Z., EICKEMEYER, R., EISEN, L., GRISWELL, J., BUYUKTOSUNOGLU, A., ZYUBAN, V., LOGAN, D., SINHAROY, B., AND TENDLER, J. 2005. Stretching the limits of clock-gating efficiency in server-class processors. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*.

KIM, N., FLAUTNER, K., BLAAUW, D., AND MUDGE, T. 2004. Single-Vdd and single-Vt super-drowsy techniques for low-leakage high-performance instruction caches. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. 2005. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro 25,* 2 (Mar.), 21–29.

LIM, C., DAASCH, W., AND CAI, G. 2002. A thermal-aware superscalar microprocessor. In *Proceedings of the International Symposium on Quality Electronic Design*.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin : building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*.

MICHAUD, P. AND SAZEIDES, Y. 2006. Scheduling issues on thermally constrained processors. Tech. Rep. PI-1822, IRISA. Also published as INRIA report RR-6006.

MICHAUD, P., SAZEIDES, Y., SEZNEC, A., CONSTANTINOU, T., AND FETIS, D. 2005. An analytical model of temperature in microprocessors. Tech. Rep. PI-1760, IRISA. Also published as INRIA report RR-5744.

MOORE, J., SHARMA, R., SHIH, R., CHASE, J., PATEL, C., AND RANGANATHAN, P. 2004. Going beyond CPUs: The potential for temperature-aware data centers. In *Proceedings of the First Workshop on Temperature-Aware Computer Systems*.

NAFFZIGER, S., STACKHOUSE, B., AND GRUTKOWSKI, T. 2005. The implementation of a 2-core multithreaded Itanium-family processor. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*.

NII, K., MAKINO, H., TUJIHASHI, Y., MORISHIMA, C., HAYAKAWA, Y., NUNOGAMI, H., ARAKAWA, T., AND HAMANO, H. 1998. A low power SRAM using auto-backgate-controlled MT-CMOS. In *Proceedings of the International Symposium on Low Power Electronics and Design*.

PHAM, D., BEHNEN, E., BOLLIGER, M., HOFSTEE, H. P., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., LE, B., MASUBUCHI, Y., POSLUSZNY, S., RILEY, M., SUZUOKI, M., WANG, M., WARNOCK, J., WEITZEL,

S., WENDEL, D., AND YAZAWA, K. 2005. The design methodology and implementation of a first-generation CELL processor: A multi-core SoC. In *Proceedings of the IEEE 2005 Custom Integrated Circuits Conference*.

POIRIER, C., MCGOWEN, R., BOSTAK, C., AND NAFFZIGER, S. 2005. Power and temperature control on a 90nm Itanium-family processor. In *EEE International Solid-State Circuits Conference Digest of Technical Papers*.

POWELL, M., GOMAA, M., AND VIJAYKUMAR, T. 2004. 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*.

ROHOU, E. AND SMITH, M. 1999. Dynamically managing processor temperature and power. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimization*.

ROTEM, E., NAVEH, A., MOFFIE, M., AND MENDELSON, A. 2004. Analysis of thermal monitor features of the Intel Pentium M processor. In *First Workshop on Temperature-Aware Computer Systems*.

SAMSON, E., MACHIROUTU, S., CHANG, J.-Y., SANTOS, I., HERMERDING, J., DANI, A., PRASHER, R., AND SONG, D. 2005. Interface material selection and a thermal management technique in second-generation platforms built on Intel Centrino Mobile Technology. *Intel Technology Journal 9,* 1 (Feb.).

SCHMIDT, R., CRUZ, E., AND LYENGAR, M. 2005. Challenges of data center thermal management. *IBM Journal of Research and Development 49,* 4/5 (July), 533–539.

SHAYESTEH, A., KURSUN, E., SHERWOOD, T., SAIR, S., AND REINMAN, G. 2005. Reducing the latency and area cost of core swapping through shared helper engines. In *Proceedings of the International Conference on Computer Design*.

SKADRON, K., STAN, M., HUANG, W., VELUSAMY, S., SANKARANARAYANAN, K., AND TARJAN, D. 2003. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*.

SRINIVASAN, J. AND ADVE, S. 2005. The importance of heat-sink modeling for DTM. In *Proceedings of the 4th Annual Workshop on Duplicating, Deconstructing, and Debunking*.

TENDLER, J., DODSON, J., FIELD, J., LE, H., AND SINHAROY, B. 2002. POWER4 system architecture. *IBM Journal of Research and Development 46,* 1 (Jan.).

TSCHANZ, J., NARENDRA, S., YE, Y., BLOECHEL, B., BORKAR, S., AND DE, V. 2003. Dynamic sleep transistor and body bias for active leakage power control of microprocessors. *IEEE Journal of Solid-State Circuits 38,* 11 (Nov.), 1838–1845.

Received March 2006; revised September 2006; accepted November 2006