Thermal Balancing Policy for Streaming Computing on Multiprocessor Architectures

Fabrizio Mulas, Michele Pittau,
Marco Buttu, Salvatore Carta,
DMI-University of Cagliari
32 Via Ospedale, Cagliari, Italy
[mulas,pittau,buttu,salvatore]@unica.it

Andrea Acquaviva
DI-University of Verona
Strada le Grazie 15, Verona, Italy
andrea.acquaviva@univr.it

Luca Benini
DEIS - University of Bologna
V.le Risorgimento 2, Bologna, Italy
lbenini@deis.unibo.it

David Atienza,
Giovanni De Micheli
LSI-EPFL
Lausanne, CH
[david.atienza,giovanni.demicheli]@epfl.ch

ABSTRACT

As feature sizes decrease, power dissipation and heat generation density exponentially increase. Thus, temperature gradients in Multiprocessor Systems on Chip (MPSoCs) can seriously impact system performance and reliability. Thermal balancing policies based on task migration have been proposed to modulate power distribution between processing cores to achieve temperature flattening. However, in the context of MPSoC for multimedia streaming computing, where timeliness is critical, the impact of migration on quality of service must be carefully analyzed. In this paper we present the design and implementation of a lightweight thermal balancing policy that reduces on-chip temperature gradients via task migration. This policy exploits run-time temperature and load information to balance the chip temperature. Moreover, we assess the effectiveness of the proposed policy for streaming computing architectures using a cycle-accurate thermal-aware emulation infrastructure. Our results using a real-life software defined radio multitask benchmark show that our policy achieves thermal balancing while keeping migration costs bounded.

1. INTRODUCTION

Power density increase and thermal management are two of the key factors limiting the performance and high-performance multicore and *Multi-Processor System-on-Chip (MPSoC)* architectures [13]. Moreover, it has been demonstrated that large temperature variations cause low reliability and they also have a negative effect on leakage [13]. Thermal balancing does not come as a side effect of energy and load balancing; thus, thermal management and balancing policies are needed [12, 5]. Although task and thread migration have been proposed to prevent thermal runaway and to achieve thermal balancing in multithreaded architectures [3, 5], in the case of MPSoC-based streaming computing architectures, which are tightly timing constrained, the cost constraints are drastically different. In this context, it is critical to evaluate thermal policies and task migration costs to provide accurate analyses of the implications of thermal gradients in processing cores, and the quality-of-service degradation due to task deadline misses. Hence, this kind of analysis needs to be performed with suitable thermal-aware simulation or emulation infrastructures, which provides the required accuracy without impacting emulation time [2]. Moreover, these infrastructures must include the impact of the software layers needed to support task migration, namely, the *Multi-Processor Operating Systems (MPOS)*, the middleware and the communication library.

In this paper we present a thermal balancing policy whose purpose is to keep the spatial and temporal temperature gradient controlled for streaming computing MPSoC architectures. The contributions of the paper are the following: (i) design of a lightweight thermal balancing policy with a fully-functional implementation; (ii) a detailed and exhaustive evaluation of task migration effects due to the application of the thermal balancing policy, allowed by the thermal-aware emulation infrastructure; and (iii) the comparison with other thermal balancing policies as well as with energy and load balancing policies using a streaming multimedia benchmark, i.e., a Software Defined FM Radio application. Our results show that our policy achieves thermal balancing with less cost in *Quality-of-Service (QoS)* than other state-of-the-art techniques, while keeping task migration overhead to a very low cost, both in performance and temperature.

This paper is organized as follows. In Section 2, we overview related work on thermal and power management techniques. In Section 3 we present our thermal balancing policy. In Section 4 we describe the complete MPOS-MPSoC emulation framework used to compare our proposed thermal balancing policy with state-of-the-art solutions to this problem. Then, in Section 5, we present our experimental results. Finally, in Section 6, we summarize the main conclusions of the paper.

2. RELATED WORK

Many recent works in computer architecture focus on power management and thermal control for multi-core and MPSoCs [9, 5, 4]. While reducing power density has the effect of reducing overall

temperature, power-aware design does not directly imply that thermal gradients between different components are minimized or individual hot spots do not appear [13, 5].

Using several high-level architectural models, related works exist that focus on the design of control policies for thermal management. [5, 1] have proposed adaptive mechanisms for thermal management, but they use techniques of a primarily power-aware nature focusing on key micro-architectural hotspots rather than mitigating thermal gradients. Also, task and thread migration techniques have been suggested in multi-core platforms. [3, 4] describe techniques for thread assignment and migration using performance counter-based information or compile-time pre-characterization. However, their techniques mainly apply to multi-threaded architectures and the assumed performance cost of thread migration are high for MPSoC and streaming applications. Finally, [11] studies the thermal behavior of low-power MPSoCs. This work concludes that for such low-power architectures, no thermal issues presently exist. However, this analysis is only applicable to very low-power embedded architectures, which have a limited processing power that is not sufficient to fulfill the requirements of the MPSoC streaming processing architectures that we cover in this work. Also, [16] investigates both power- and thermal-aware techniques for task allocation and scheduling. This work shows that thermal-aware approaches outperforms power-aware schemes in terms of maximal and average temperature reductions. However, it does not come up with a thermal balancing algorithm as we present in this paper.

To fully assess the effectiveness of these techniques, an accurate thermal-aware simulation/emulation framework is needed. Several groups have proposed schemes for thermal modeling and simulation at different levels of abstraction. [13] presents a thermal/power model for super-scalar architectures. [14] outlines a simulation model on embedded cores, which show thermal gradients across the cores. [8] explores methods to model performance and power efficiency for multicore processors, but they do not study thermal management policies.

3. THERMAL BALANCING POLICY

The development of thermal balancing policies is needed because temperature gradients affect reliability and leakage. In general, thermal balancing does not come as a side effect of energy balancing. In Figure 1.a a typical situation where a two core system running three tasks (A, B, C) is energy balanced but thermally unbalanced is shown. Both processors can independently set their frequency and voltage to reduce energy/power dissipation to the minimum required by the current load. Tasks are characterized by their full-speed-equivalent load (FSE), that is the load imposed by a task when the core runs at full speed (maximum available frequency). Core 1 runs tasks A and B, having FSE of 50% and 40% respectively; core 2 runs task C that has a FSE of 40%. In this case core 1 can ideally scale its frequency to 90% of its maximum value, while core 2 can scale it to 40%. No better tasks mapping exists that further reduces energy/power dissipation. In this situation, due to the different power consumed, temperature of core 1 will be higher than temperature of core 2. As such, a thermally balanced condition can be achieved by periodically migrating task B from the first core to the second core [16] (as represented in Figure 1.b), obtaining, on average, an equalized workload on the two cores (i.e., 40% +50%/2 = 65%). If the temperature variations caused by migrations are slower than the migration period, a temperature close to the average workload (i.e, 65%) will be achieved on both cores. Clearly, this is a simplified case. The challenge of a thermal balancing algorithm is the selection of the task sets to be migrated back and forth between two or more cores so that temperature is balanced while keeping migration costs bounded.

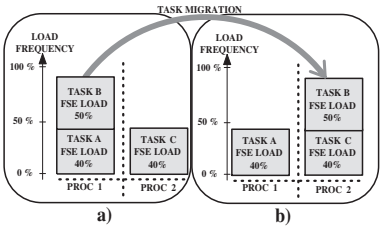


Figure 1: Simple thermal balancing example

In this section we first describe the thermal balancing algorithm, then we discuss the implementation of the task migration infrastructure needed to support the policy implementation.

3.1 Thermal Balancing Algorithm

The thermal balancing strategy we propose in this paper is inspired by MiGra [2]. To prevent impact on QoS caused by migration, MiGra is based on run-time estimation of migration costs to filter migration requests driven by temperature differences between cores. We tuned MiGra by including migration costs caused by the actual migration infrastructure. These costs, as explained later in this section, are mainly due to data transfer and synchronization between migration daemons. Moreover, in our implementation MiGra lies on top of a dynamic voltage/frequency scaling (DVFS) policy [5]. Thus, the power consumption of a task is proportional to its load.

The rationale of the strategy we implemented is to bound the temperature of each processor around the current mean temperature, trying to minimize the overhead in terms of number of migrated tasks and amount of data transferred due to migrations. A maximum distance of the temperature of each processor from the current average temperature is defined, identifying a range of allowed temperatures for each single processor between an upper and a lower threshold. It must be noted that these thresholds are not related to the temperature limit defined by the system designer to prevent thermal runaway. Note that thermal runaway can be managed by using stopping the core when it reaches a temperature above a predefined panic threshold. The operating point of our strategy is below this threshold because it is aimed at reducing temperature gradients.

Each time the temperature of a processor reaches the upper threshold, a migration is triggered so that a set of tasks is moved away from that processor to another processor having a temperature below the current average temperature. On the other side, each time the temperature of a processor reaches the lower threshold, a migration is triggered so that a set of tasks is moved away from that processor to another processor having a temperature above the current average temperature.

To reduce the amount of computation needed to select the tasks to move, the algorithm moves tasks only between two processors at a time. This means that the processor that triggers the migration (e.g. a hot one) will select only one target processor (e.g. a cold one) in order to balance the workload between them. A fundamental constraint of the thermal balancing strategy we propose is that it reduces thermal gradient without impacting energy dissipation.

The algorithm consists of two phases. In first phase the candidate processors (source and target) are selected, while in the second phase the task sets to be exchanged are defined. The first phase works as follows. If all the following three conditions are verified dst processor becomes a candidate to exchange workload with src processor:

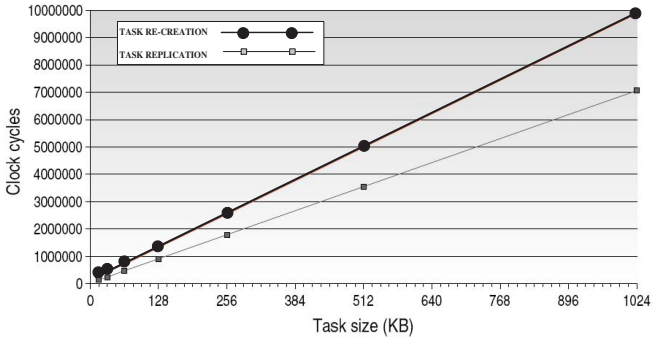


Figure 2: Migration cost as a function of task size for task-replication and task-recreation.

- If the source is warm, the destination processor has to be cold, and viceversa: $(t_{src} - t_{mean}) * (t_{dst} - t_{mean}) < 0$
- If the frequency of the source processor is above the threshold frequency, the frequency of the destination processor has to be below it, and viceversa: $(f_{src} - f_{mean}) * (f_{dst} - f_{mean}) < 0$
- The total power wasted by the two processors after the migration has to be lower than the total power wasted by the two processors before the migration: $({f^2}_{src} + {f^2}_{dst})_{before\_migr} \geq ({f^2}_{src} + {f^2}_{dst})_{after\_migr}$

The selection of the number of tasks and the final target processor depends on the evaluation of migration costs. We considered a cost function which is the product of the amount of data moved due to the migration times the frequency of migrations. To estimate the migration frequency we can consider that, for a given temperature difference between two processors, the need of triggering a new migration is proportional to the difference between the current temperature of the processor target of the migration and the average temperature of the chip. The target processor (tgt) of the migration is selected using equation 1:

$$tgt : \frac{\sum_i^I ({C^{src}}_i) + \sum_j^J ({C^{tgt}}_j))}{(t_{tgt} - t_{mean})^2} \equiv MIN \qquad (1)$$

Where ${C^{src}}_i$ is the amount of data to move for the $i-th$ of I tasks running on the source processor, ${C^{tgt}}_j$ is the amount of data to move for the $j-th$ of J tasks running on the tgt processor.

An exhaustive approach where a full search is performed by comparing the migration costs of all the possible combinations of tasks is not practical. For this reason, a reasonable approximation can be introduced, by considering the effect of migration of a task on the temperature balancing decreases together with its load. This leads to the fact that we can limit the number of tasks to be considered only to the few tasks having the highest load.

3.2 Middleware Support for Task Migration

In this work we focus on a homogeneous architecture such as the one shown in Figure 3.a. The architectural template we consider is based on 32-bit RISC processors without memory management unit (MMU) accessing cacheable private memories and a single non-cacheable shared memory. This architecture addresses the scalability issues proper of symmetric multiprocessors that limit the number of integrable cores. It follows the structure envisioned for non-cache-coherent MPSoCs [7, 15].

On the software side, each core runs its own instance of the uClinux OS [10] in the private memory. The uClinux OS is a derivative of Linux 2.4 kernel intended for microcontrollers without MMU. Each task is represented using the process abstraction, having its own private address space. As a consequence, communication has to be explicitly carried on using a dedicated shared memory area on the same on-chip bus. The OS running on each core sees the shared area as an external memory space.

The software abstraction layer is described in Figure 3.b. Since uClinux is natively designed for single-processor environments, we added the support for interprocessor communication in the middleware. Then, on top of the local OSes we developed a layered software infrastructure to provide and efficient parallel programming model for MPSoC-MPOS software developers, including a task migration support layer.

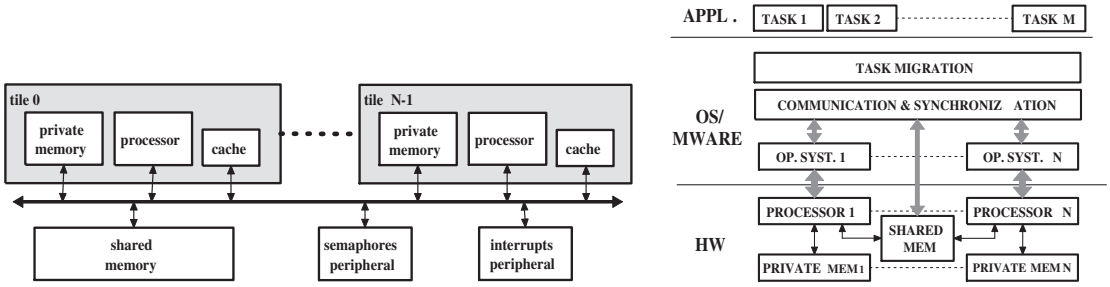
Task Migration Support

In our implementation, migration is allowed only at predefined checkpoints, that are provided to the user through a library of functions together with message passing primitives. A so called *master daemon* runs in one of the cores and takes care of dispatching tasks on the processors.

We implemented a migration mechanisms that differs in the way the memory is managed. A first version, based on a *task-recreation* strategy, which kills the process on the original processor and recreates it from scratch on the target processor. This strategy only works in operating systems supporting dynamic loading, such as uClinux. Task recreation is based on the execution of fork-exec system calls that takes care of allocating the memory space required for the incoming task. To support task recreation on an architecture without MMU performing hardware address translation, a position independent type of code (called PIC) is required to prevent the generation of wrong references of pointers, since the starting address of the process memory space may change upon migration. Unfortunately, PIC is not supported by the target processor we are using in our platform (microblazes) [17].

Due to the above limitation, we implemented an alternative migration strategy where a replica of each task is present in each local OS, called *task-replication*. Only one processor at a time can run one replica of the task. While in one processor the task is executed normally, in the other processors it is in a queue of suspended tasks. As such, a memory area is reserved for each replica in the local memory, while kernel-level task-related information are allocated by each OS in the Process Control Block (PCB) (i.e. an array of pointers to the resources of the task). Therefore, task replication is suitable for deeply embedded operating systems without dynamic loading because the absolute memory position of the process address space does not change upon migration, since it can be statically allocated at compile time. In fact, even if this technique leads to a waste of memory for migratable tasks, it has the advantage of being faster, since it cuts down on memory allocation time with respect to a task recreation.

A quantification of the memory overhead due to task replication is shown in Figure 2. In this figure, the costs is shown in terms of processor cycles needed to perform a migration as a function of the task size. In both cases, part of the migration overhead is due to the amount of data transferred through the shared memory. Moreover, for the task recreation technique, there is another overhead due to the additional time required to re-load the program code from the file system; thus, the offset that appears between the two curves. Moreover, the task recreation curve has a larger slope due to the large amount of memory transfers, which leads to an increasing contention on the bus. Hence, the contribution on the execution time increases as file size increases in comparison to the task repli-



a) b)

Figure 3: a) Target hardware architecture; b) Scheme of the software abstraction layer.

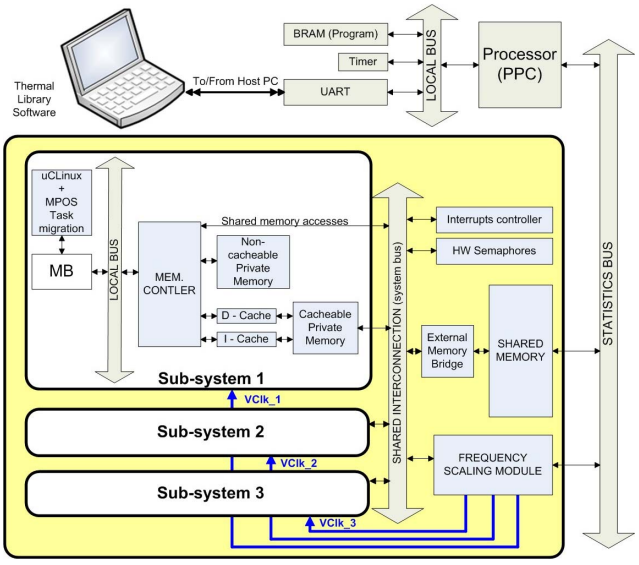


Figure 4: Overview thermal emulation framework

cation case.

During execution, when a task reaches a user-defined checkpoint, it checks for migration requests performed by the master daemon. If the migration is taken, the task is either suspended or killed (depending on the strategy); thus, waiting to be deallocated and restored on another processor from the migration middleware. When the master daemon decides to migrate a task, it signals to the slave daemons of the source processor the migration of the task. Then, a dedicated shared memory space is used as a buffer for the task context transfer. To assist migration decision, each slave daemon writes in a shared data structure the statistics related to local task execution (e.g. processor utilization and memory occupation of each task), which are periodically read by the master daemon.

4. THERMAL EMULATION FRAMEWORK

Our thermal balancing strategy needs to be evaluated in realistic MPSoC-MPOS architectures. For this evaluation we need cycle-accurate models to extract detail statistics for hardware components, operating system and middleware operations, while long real-life workloads from streaming applications are executed. Thus, in this work we have built a complete estimation framework extending the flexible HW/SW FPGA-based hardware emulation infrastructure presented in [2]. An overview of the whole current framework built in this work is presented in Figure 4. Using this framework, we can can extract thermal statistics from the processors, I-cache and D-caches and external memory accesses.

The system can be scaled to any number of cores sub-systems

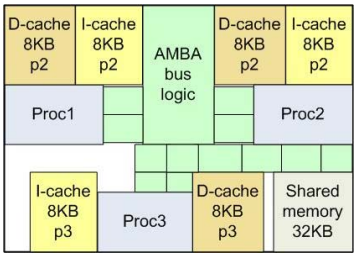


Figure 5: Emulated MPSoC floorplan

by using appropriate FPGAs, in this work we used a Virtex-2 Pro vp30 board [17]. Then, as Figure 4 shows, a specialized thermal monitoring subsystem is included using hardware sniffers, a virtual clock management peripheral and a dedicated Power PC that implements the extraction of statistics through a UART port. In Table 1, we summarize the values used for the components of our emulated MPSoC. These values have been derived from industrial power models for a 0.09 μm CMOS technology. The energy figures for each component are provided to a software thermal library running on a host PC, which is based on the Hotspot thermal analysis tool [13]. This library calculates the temperature of each tridimensional cell of the emulated MPSoC floorplan (Figure 5) and the temperature of each processor is visible through shared memory locations for our uClinux-based MPOS, which applies the thermal balancing policies to the system (see Section 5 for more details). The update of memory locations with thermal figures occurs every 10 ms to guarantee accurate thermal monitoring.

Using Floorplan 5, we compare two different thermal packages. The first packaging solution was derived from real-life streaming SoCs [6] for mobile embedded targets, where temperature rising of around 10 degrees Centigrades requires few seconds to take place. In addition, our second packaging solution models highly variant (i.e., high-performance) SoCs from the thermal gradient viewpoint [13]; thus, significant temperature rising effects can occur in less than a second. Our policy is validated against both packaging solutions.

Table 1: Power of components in 0.09 μm CMOS

	Max. Power@500 MHz
RISC32-streaming (Conf1)	0.5W (Max)
RISC32-ARM11 (Conf2)	0.27W (Max)
DCache 8kB/2way	43mW
ICache 8kB/DM	11mW
Memory 32kB	15mW

5. EXPERIMENTAL RESULTS

We have validated our policy onto a 3-core MPSoC where a multi-task streaming application is currently mapped. The experiments were conducted considering the power figures and two dif-

ferent packaging models described in Section 4. The metrics we have studied are: i) Spatial and temporal variance of the temperatures of the processors; ii) Average quantity of migrated data and number of migrated tasks; iii) QoS degradation as the percentage of frame miss rate. The results are compared with an energy-aware balancing and a Stop&Go policy [5], which controls processor temperature without exploiting task migration.

5.1 Benchmark application description

To evaluate the effectiveness of the proposed policy we ported to our system a *Software FM Defined Radio (SDR)* benchmark, which is representative of a large class of streaming multimedia applications.

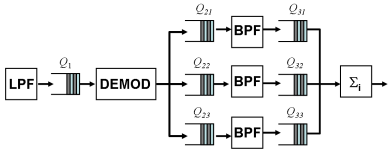


Figure 6: SDR case study

As shown in Figure 6, the application is composed by various tasks, graphically represented as blocks. Input data represent samples of the digitalized PCM radio signal which has to be processed in order to produce an equalized base-band audio signal. In the first step, the radio signal passes through a *Low-Pass-Filter* (LPF) to cut frequencies over the radio bandwidth. Then, it is demodulated by the *demodulator* (DEMOD) to shift the signal at the baseband and produce the audio signal. The audio signal is then equalized with a number of *Band-Pass-Filters* (BPF) implemented with a parallel structure. Finally, the *consumer* (Σ) collects the data provided by each BPF and makes the sum with different weights (gains) in order to produce the final output. Communication among tasks is done using message queues, each task reads data from its input queue and sends the results to the output queue, where the next task in the software pipeline reads them.

5.2 Policy evaluation and comparison

The following policies were applied on the SDR benchmark:

Energy-Balancing. This policy maps the tasks of the SDR application such as their energy consumption is balanced [1] among the cores. Energy is computed from the frequency and voltage imposed by the tasks running, which are dynamically adjusted using a DVFS algorithm [5].

Stop&Go. This policy prevents thermal runaway, i.e., it shuts down a core when it reaches a panic temperature threshold. In its original version [5], core execution is resumed after a predefined timeout. We modified this policy to fairly compare it with our thermal balancing algorithm by using the upper threshold of our algorithm as the panic threshold, and our lower threshold to define when to switch the core on instead of timing out.

Migration-based Thermal Balancing. This is our proposed policy in this paper. To perform a fair comparison between the policies, we started from a statically energy-balanced configuration for both Stop&Go and our policy. This configuration is described in Table 2 where names, loads and running frequencies for each task are detailed. Our goal is to demonstrate that thermal balancing reduces thermal gradients in comparison to an already energy-balanced condition without impacting QoS.

Mobile embedded targets.

In the first set of experiments we evaluate the behavior of our policy. To this end, we tested it in our emulation platform using the 3-core configuration and thermal package derived from 90nm cores [6] for real-life streaming SoCs [6] (Section 4). After a first execution

Core / freq.	*Task*	*Load* [%]
Core 1 (533 MHz)	BPF1	36,7
	DEMOD	28,3
Core 2 (266 MHz)	BPF2	60,9
	Σ	6,2
Core 3 (266 MHz)	BPF3	60,9
	LPF	18,8

Table 2: Application mapping

phase (12.5 sec), the temperatures of the three cores gets stable. However, the temperature is not balanced, i.e. 10 degrees Centigrades exist between the hottest (core 1) and the coolest core (core 3). This thermal configuration is due to the application of DVFS to each core. Moreover, although core 2 and 3 have the same frequency, their temperatures differ because of the different heat spreading capabilities due to their position in the floorplan 5. Thus, in our experiments, we trigger our task-migration based policy to achieve thermal balancing after this initial phase.

When thermal balancing is applied, each time a core reaches the upper threshold (set to $\pm$ 3 degrees more than the average temperature), a migration is triggered, one task is moved to a colder core, and the temperature becomes balanced for all cores within 1 second of execution of the SDR application. Thus, showing the efficacy of our policy to reach temperature balancing in streaming SoCs [6]. Moreover, our results indicate that the temperature of the hottest core passes the upper threshold while balancing the temperature for a very limited time (less than 400 ms).

A quantitative evaluation and comparison among the three policies is provided in the following experiments for the same packaging configuration. Figure 7 shows the temperature standard deviation for the three policies as a function of the threshold values. The X-axis indicates the distance of upper and lower threshold from the mean temperature. As this figure shows, the temperature deviation increases with the threshold. Thus, our policy is more effective in reducing temperature deviation than other techniques because it acts on both hot and cold cores. In particular, Stop&Go does not change the temperature of the cold cores.

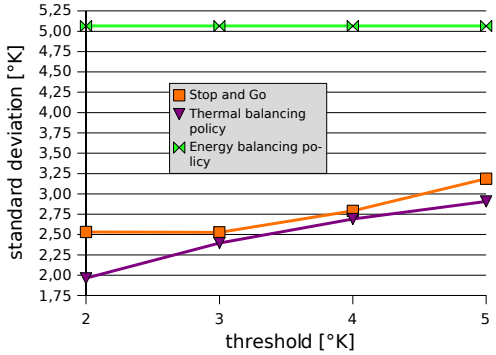


Figure 7: Temp. standard deviation for embedded SoCs

Then, Figure 8 shows the number of deadline misses as a function of the threshold values. As shown, our policy leads to few deadline misses while Stop&Go suffers a higher value of missed frames. Deadline misses may be caused by frozen tasks during migration; hence, interprocessor queues are depleted during migration, and if the queue of the last stage gets empty a deadline miss occurs. However, as Figure 8 illustrates, migration is lightweight and fast enough to limit this drawback. In fact, missed frames appear only for the minimum threshold considered in our experiments. Furthermore, we observed that the average queue level does not change because of migration; thus, a queue size handling ther-

mal balancing can always be found and the SDF application can sustain thermal balancing without QoS impact, i.e., the minimum queue size to sustain migration in our experiments was 11 frames. Hence, this buffering size is feasible for streaming applications.

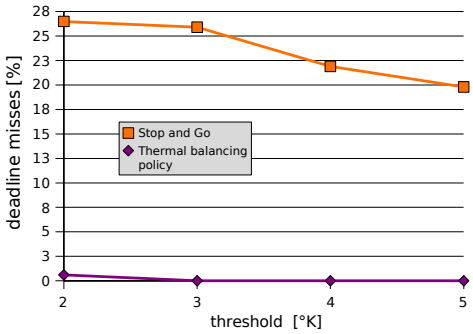


Figure 8: Deadline misses for the embedded mobile system

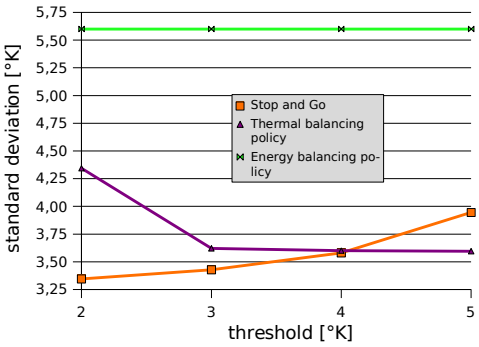


Figure 9: Standard deviation the high performance SoCs

High-performance embedded targets.

To stress our policy when temperature variations are faster, we repeated our experiments using the alternative packaging value for high-performance systems (see Section 4), where temperature variations are 6× faster than the previous model.

Figure 9 shows the standard deviation of the temperature for the three policies. The energy balancing policies achieve very poor results and the modified Stop&Go policy behaves better in terms of temperature deviation, but it causes a large amount of deadline misses (Figure 10). On the contrary, our algorithm makes temperature oscillate more than Stop&Go, but causes a lot less deadline misses. Moreover, our algorithm starts behaving significantly better than Stop&Go when the threshold increases, as less migrations are triggered. Also, we observed that Stop&Go causes less deadline misses with the fast thermal model than with the slow one, due to the faster speed the lower threshold is reached after shutdown. From these experiments, we can conclude that pure software techniques cannot handle fast temperature variations, and a hardware-software co-design approach is needed.

Finally, Figure 11 depicts the average number of migrations per second performed by our algorithm for both mobile embedded and high-performance systems. As expected, the number of migrations is higher for high-performance systems. However, as each migration implies a transfer of 64 Kbytes of data (the minimum memory space allocated by the OS), the required three migrations per second are equivalent to $64*3 = 192$ Kbytes per second, which means that our task migration policy implies only a negligible overhead.

6. CONCLUSIONS

In this paper we have proposed and implemented a lightweight thermal balancing policy that reduces on-chip temperature gradients via task migration. Our validation using a software defined radio benchmark, executed on a cycle-accurate thermal-aware emulation infrastructure, has proved that this policy effectively uses run-time workload and thermal information to balance the temperature in streaming computing architectures, while keeping migration costs and deadline misses bounded.

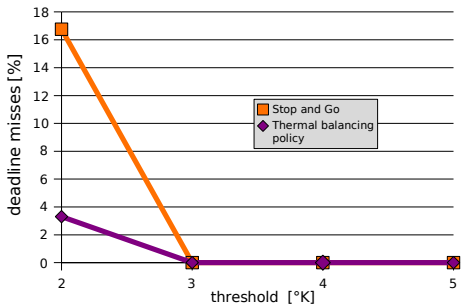


Figure 10: Deadline misses for high-performance systems

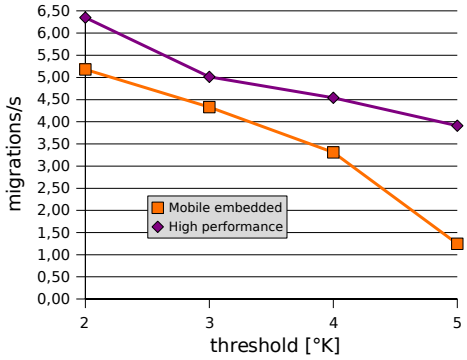


Figure 11: Migrations per sec. for both systems

7. REFERENCES

[1] F. Bellosa, et al. Event-driven energy accounting for dynamic thermal management. *Proc. COLP*, 2003.

[2] Omitted for blind review

[3] P. Chaparro, et al. Understanding the thermal implications of multi-core architectures. *TPDS*, 2007.

[4] J. Donald, et al. Power efficiency for variation-tolerant multicore processors. *Proc. ISLPED*, 2006.

[5] J. Donald, et al. Techniques for multicore thermal management: Classification and new exploration. *Proc. ISCA*, 2006.

[6] Freescale, i.mx31 multimedia applications processors, 2003. `www.freescale.com/imx31`.

[7] L. Friebe, et al. Hibrid-soc: A SoC architecture with two multimedia dsps and a risc core. *Proc. SOC*, 2003.

[8] J. Li, et al. Power-performance implications of thread-level parallelism in chip multiprocessors. *Proc. ISPASS*, 2005.

[9] R. Mukherjee, et al.. Physical aware frequency selection for dynamic thermal management in multi-core systems. *Proc. ICCAD*, 2006.

[10] uclinux: Embedded linux/microcontroller project, 2006. http://www.uclinux.org/.

[11] G. Paci, et al. Exploring temperature-aware design in low-power MPSoCs. *Proc. DATE*, 2006.

[12] T. Sato, et al. On-chip thermal gradient analysis and temperature flattening for soc design. *Proc. ASP-DAC*,2005.

[13] K. Skadron, et al. Temperature-aware microarchitecture: Modeling and implementation. *ACM TACO*, 2004.

[14] H. Su, et al. Full chip leakage estimation considering power supply and temperature variations. *Proc. ISLPED*, 2003.

[15] P. van der Wolf, et al. Design and programming of embedded multiprocessors: an interface-centric approach. *Proc. CODES+ISSS*, 2004.

[16] Y. Xie, et al.. Temperature-aware task allocation and scheduling for embedded MPSoC design. *J-VLSI SPS*, 2006.

[17] Xilinx, XUP Virtex-II Pro development system, 2006. `http://www.xilinx.com/univ/xupv2p.html`.