Multi-Processor Operating System Emulation Framework with Thermal Feedback for Systems-on-Chip [*]

Salvatore Carta, Michele Pittau
DMI-University Cagliari, Italy.

Andrea Acquaviva,
STI/University of Urbino, Italy.

Pablo G. Del Valle[†], David Atienza[†], Giovanni De Micheli
LSI/EPFL, Switzerland.

Fernando Rincon
UCLM, Ciudad Real, Spain.

Luca Benini
DEIS/Bologna University, Italy.

Jose M. Mendias
[†]DACYA/Complutense University of Madrid, Spain.

ABSTRACT

Multi-Processor System-On-Chip (MPSoC) can provide the performance levels required by high-end embedded applications. However, they do so at the price of an increasing power density, which may lead to thermal runaway if coupled with low-cost packaging and cooling. Hence, mechanisms to efficiently evaluate the effectiveness of advanced thermal-aware operating-system (OS) strategies (e.g. task migration) onto the available MPSoC hardware are needed.

In this paper, we propose a new MPSoC OS emulation framework that enables the study of thermal management strategies at the architectural- and OS-levels with the help of a standard FPGA. This framework includes the hardware and software components needed to accurately model complex MPSoCs architectures, and to test the effects of run-time thermal management strategies at the OS/middleware level with real-life inputs. Our results show that migration overhead is negligible w.r.t. temperature timings, enabling the development of thermal-aware migration strategies. Moreover, the effectiveness of the monitoring and feedback mechanism provides an emulation performance only ten times slower than real time.

General Terms: Design, Measurement, Performance.
Keywords: Operating System, MPSoC, Thermal Studies, FPGA, Emulation.

1. INTRODUCTION

The ever increasing complexity of consumer applications (e.g. multimedia processing or 3D games) for portable devices such as smart-phones and palmtop computers demands complex hardware and software designs to meet tight performance requirements while respecting power constraints. In this context, MULTI-PROCESSOR SYSTEMS-ON-CHIPS (MPSOCS) have been proposed as a promising solution [9]. Multi-processor operating systems (MPOS) and middleware are required to efficiently exploit the interaction of the various components of the underlying hardware, while ensuring flexibility and providing a standard hardware-abstraction layer for heterogenous application development.

While this layered approach eases the programmer's job, software and hardware designers have the responsibility of efficiently managing non-functional system constraints, such as power and temperature. In fact, it has been recently shown [17] that MPSoCs can experience hotspots and very high temperatures in forthcoming technology nodes. It is evident that the high hardware and software complexity provides high degree of freedom at the price of increased design effort at operating system (OS) and middleware level. For this reason, new methods that allow designers to test thermal-aware MPOS strategies (e.g. voltage scaling and task migration) on MPSoCs architectures early in the system integration flow need to be developed to match time-to-market requirements.

Presently, a few of cycle-accurate MPSoC simulators have been developed that include MPOS support [4]. However, these simulators are inappropriate to perform long thermal simulations due to their limited performance (circa 10-100 Khz). Also, other higher abstraction levels simulators (e.g. at the transaction level) provide faster simulations, but the accuracy during the evaluation of thermal effects is limited. An alternative to cycle-accurate simulators for thermal evaluation is MPSoC hardware emulation [3, 5, 2, 6]. However, most emulation frameworks are only limited to emulation of the behavior of pure hardware components of MPSoC architectures. Moreover, FPGA vendors only include support for monoprocessor OSes, but no frameworks are available to perform thermal exploration of MPOS behavior with the underlying MPSoC architecture.

In this paper we present a new MPOS emulation framework for MPSoC that enables the study of thermal management strategies at the architectural- and OS-levels using a standard FPGA. We have developed within this framework the necessary hardware and software extensions to allow designers to test different thermal-aware MPOS implementations running onto real-life MPSoC architectures emulated on FPGAs. To the best of our knowledge, this is the first multiprocessor platform that supports OS and middleware emulation, and enables the exploration of closed-loop policies that dynamically adjust system operation based on monitoring of die temperature. Our results show the benefits of advanced temper-

[*]This work is partially supported by the Spanish Government Research Grants TIN2005-5619 and FPU AP2005-0073, and the Swiss NSF Research Grant 20021-109450/1.

ature management exploiting OS facilities like task migration in MPSoCs.

The paper is organized as follows. In Section 2, we overview related work on MPOS design and MPSoC validation. In Section 3 we present the architectural extensions to MPSoC designs to provide an efficient implementation of MPOSes. In Section 4 we describe the foundations of the ported MPOS to enable a complete framework to explore thermal-aware OS-level strategies. In Section 5, we detail the complete MPOS MPSoC emulation flow. In Section 6, we present our experimental results. Finally, Section 7 summarizes the contributions of the paper and presents possible future research directions.

2. RELATED WORK

In the last years research on suitable modeling and design tools for MPSoCs platforms to run consumer applications have started to be proposed [9, 14, 4]. This research effort includes both hardware and software approaches that address the problem of providing exploration and validation methods for MPSoCs.

Hardware prototyping has already been used in industry for several years as a good validation method for industrial MPSoCs. In this regard, Palladium II [5], Zebu-XL [6] and System Explore [1] have been proposed. Nevertheless, they typically have a high cost (more than \$300K-\$400K) and operate in the order of few MHzs. Then, Heron [7] and ASIC Integrator [2] are faster for MPSoCs architectural exploration, but they are limited to using few proprietary cores (e.g. AMBA interconnects and few ARM-based cores in ASIC Integrator).

Similarly, recent MPSoC emulation platforms have been recently proposed in the academic context. In [14] it is presented a framework that enables designers to explore different interconnection mechanisms of MPSoC architectures including several proprietary 32-bit VLIW cores. Also, [13] describes a combined hardware-software method that can speed up software simulators with FPGA emulation, by synchronizing both sides in a cycle-by-cycle basis using a shared register bank. This work shows a final speed for the combined hardware-software framework of 1 MHz. However, none of these works include a ported OS and thermal modelling combined with the emulation of MPSoC architectures.

The RAMP (Research Accelerator for Multiprocessors) [11] project exploits a hardware-software infrastructure close to the one used in our work. Multiple operating systems run on a emulated multiprocessor hardware. There are two main points of distinction compared to this work: i) we developed a middleware infrastructure supporting basic communication and synchronization but also advanced services such as task migration between processing elements; ii) we implemented a statistics collection support that interfaces with a thermal model to develop and test strategies for closed-loop thermal control.

Hot spots and thermal modeling in general is a very important concern in latest multi-processors [19, 10], and temperature-aware design and tools to support it are in great need. In [17] it is proposed a thermal software model to predict the temperature rise effects in the different components of super-scalar microarchitectures (e.g. performance degradation, increase in leakage power, etc). Also, in [18] it is studied temperature and voltage variations in embedded cores, which shows variations of 13.6 degrees across the die. Finally, regarding thermal hardware emulation, in [12] it is explored the use of ring-oscillators, which can dynamically be inserted or eliminated, for thermal monitoring in FPGA-based embedded designs. This empirical measurement method is interesting, but only applicable to FPGAs as target devices, while the proposed general emulation framework is able to model the temperature of MPSoC designs implemented with ICs and running MPOSes.

With respect to the work presented in [3] we improved it in several points. First, we added a more flexible clock management that allow an independent runtime frequency scaling support for each processor. Second, we implemented the hardware support needed to support OS and middleware with interprocessor task communication and synchronization.

To summarize, with respect to the state of the art, thanks to our emulation system it is possible for the first time to develop thermal aware strategies at the middleware and OS level before the system prototype is available, without compromising cycle-level accuracy.

3. MPOS MPSOC FPGA EMULATION AND THERMAL MONITORING

The proposed MPSoC framework exploits FPGA emulation to model the hardware components of the considered MPSoC platform at multi-megahertz speeds. The hardware architecture is composed of a variable number of soft-cores (currently up to four cores). Each core runs from the private memory its own instance of a customized version of uClinux operating system [15] that has been ported and optimized for the underlying hardware. UClinux is an operating system that includes a collection of Linux 2.x kernel releases intended for single microcontrollers without Memory Management Units (MMUs), as well as a collection of user applications and libraries. Next, a shared memory is used by a middleware layer running on top of each OS for communication, synchronization and task migration between the OSes. From the hardware viewpoint, the OSes need special support for inter-processor communication, that will be described later in Section 3.2. In Section 3.1 the basic architecture and thermal monitoring features are described.

3.1 MPSoC Basic Architecture and Thermal Monitoring

The proposed MPSoC MPOS framework is an evolution of the HW/SW FPGA-based hardware emulation infrastructure presented in [3]. An overview of the whole current framework is presented in Figure 1. Using this framework, designers can extract statistics concerning processing cores, memory subsystem and interconnection infrastructure.

In our current emulation system we can include up to four microblaze cores, due to the size of the underlying Virtex-II Pro v2vp30 FPGA. However, the system can be scaled to any number of cores by using available larger FPGAs. A specialized thermal monitoring subsystem is included. It is based on hardware sniffers, a virtual clock management peripheral and a dedicated non-intrusive subsystem (PPC subsystem in Figure1) that implements the extraction of statistics through a UART port, which are then provided to a software thermal library for bulk silicon chip systems.

The library resides in a host PC and calculates the temperature of each cell according to the floorplan of the emulated MPSoC and the frequency/voltage of each MB processor. Temperatures coming out form the library provide a real-time thermal feedback visible by the running uClinux in each processor.

The library resides in a host PC and calculates the temperature of each cell according to the floorplan of the emulated MPSoC and the frequency/voltage of each MB processor to provide real-time thermal feedback. The temperatures are then visible from the OSes and middleware through emulated memory mapped temperature sensors, which are updated by the thermal monitoring subsystem. The frequency of the regular updates of the emulated temperature sensors is configurable in the range of 10 ms to 1 s. In our experiments we have fixed this interval to 10 ms to guarantee very accurate ther-

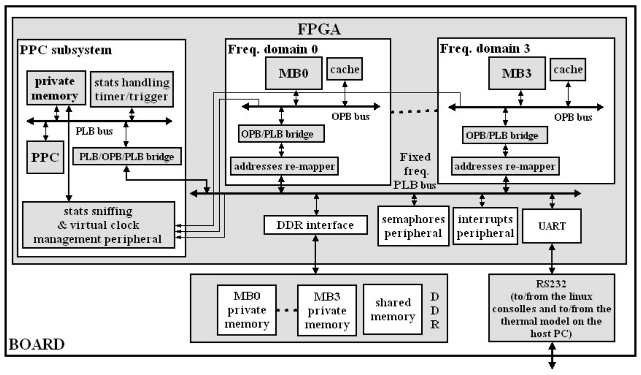

Figure 1: Overview HW architecture of emulated MPOS MPSoC platforms

mal monitoring (see Section 6).

3.2 MPOS Architectural Extensions

To support MPOSes, dedicated hardware must be designed to support OS execution and communication between process running on different processors. This includes: i) interprocessor interrupt controller; ii) semaphore memory; iii) address translator; iv) frequency scaling support. Moreover, a customized shared communication link that exploits the platform serial port has to be developed for synchronizing the console output of the OSes. Each OS runs in a private memory that is physically mapped into the available of-chip DDR memory on the board, for space reasons. In fact, the included on-chip BRAM memories of the FPGA are too small for containing the OS image. In addition, the global shared memory for the communication of OSes is also mapped into the external DDR memory.

Interprocessor interrupt controller. This component is needed to enable interrupt based wake-up of tasks sleeping while waiting for a shared resource to be released. Without interrupt support, a task can only perform busy waiting on shared variables for accessing shared data, such as messages from tasks in other processors. Interrupts can be sent to a selected processor by writing a word in a memory mapped control register.

Semaphore memory. Mutual exclusive access to the shared memory is provided through a hardware mutex implementing the test-and-set-lock (TSL) atomic operation. When reading a ”zero” value from a certain location, the value atomically becomes ”one”. The behavior of write operations is as in a normal memory afterwards. TSL is used to implement atomic wait operations on semaphores. The mutex is a memory-mapped peripheral where its lock can be acquired by any of the processors included in the emulated MPSoC. In addition, the mutex peripheral is used by every processor as a shared memory area, where other processors can deliver their messages. As such, a user-defined number of semaphores can be defined as variables into this memory. Every processor should then periodically check its shared area for new incoming messages, which would result in extra bus traffic. Therefore, to avoid this polling overhead, the mutex is able to monitor all accesses to the shared memory, and fire an interrupt for the corresponding processor only when new data is available.

Address translator. Since all the private memories are mapped in the same SDRAM, they lie in non-overlapping address ranges. Without MMU support, to avoid static linking of OS and program code at different locations, it is needed to provide to each microblaze the same view of the private memory. This is obtained by translating the addresses generated by the cores to the appropriate memory range, so that all the processes can execute independently from the processor where they run.

Frequency scaling support. Independent frequency scaling support is needed to emulate speed scaling policies. Hardware programmable dividers have been placed in the output of the platform clock generators to obtain a configurable frequency setting support. Each core can set its own frequency at run-time as well as the frequency of other cores by accessing the memory locations where the described dividers are mapped.

4. MPOS EXPLORATION FRAMEWORK

This section describes the software abstraction layer aimed to support task migration for thermal management exploration. In the programming model we adopted, each task is represented using the process abstraction. This means that each task has its own private address space. As a consequence, task communication has to be explicitly carried on using a dedicated shared memory area. For communication between tasks on the same processor, the shared memory is a shared space made available by the residing OS.

The software abstraction layer is described in Figure 2. It is based on three main components: (i) Stand alone OS for each processor running in private memory; (ii) lightweight middleware layer providing synchronization and communication services; (iii) task migration support layer.

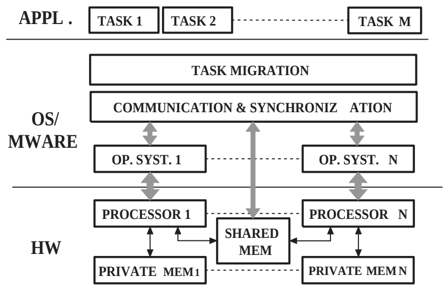

Figure 2: Scheme of the software abstraction layer

Since OSes run independently in each private memory, data can be shared between tasks using explicit services given by the underlying middleware/OS.

4.1 Communication and Synchronization Support

The communication library supports message passing through mailboxes. They are located either in the shared memory space or in smaller private scratch-pad memories, depending on their size. We implemented a lightweight message passing scheme able to exploit scratch-pad memories or physical shared memory to implement ingoing mailboxes for each processor core. We defined a library of user-level functions and system calls that each process can use to perform blocking write and read of messages on data buffers.

To use shared memory paradigm, two or more tasks are enabled to access a memory segment through a *shared malloc* that returns a pointer to the shared memory area. The implementation of this additional system call is needed because by default the OS is not aware of the external shared memory. When one task writes into a shared memory location, all the other tasks update their internal data structure to account for this modification. Allocation in shared memory is implemented using a parallel version of the Kingsley allocator, commonly used in linux kernels.

Task and OS synchronization is supported providing basic primitives like binary and counting semaphores. Both spinlock and blocking versions of semaphores are provided. Spinlock semaphores are based on hardware test-and-set memory-mapped peripherals, while non-blocking semaphores also exploit hardware inter-processor interrupts to signal waiting tasks.

4.2 Task Migration Support

To enable task migration, we implemented a task replication strategy enabled by the middleware support. A so called *master daemon* runs in one of the cores and takes care of dispatching tasks on the processors. By default, when the user launches a task, a replica of it is generated in each OS by means of a fork system call. However, only one processor at a time can run one replica of the task. While here 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 allocates by each OS in the Process Control Block (PCB) (i.e. an array of pointers to the resources of the task). Even if this technique leads to a waste of memory, it has the main advantage of being fast, since it cuts down on memory allocation time.

The migration process is managed using two kinds of kernel daemons (part of the middleware layer), a master daemon running in a single processor, and slave daemons running in all the processors. The master daemon takes care of implementing the run-time thermal-aware task allocation policy. Tasks can be migrated only corresponding to user-defined checkpoints. The code of the checkpoints is provided as a library to the programmer. When a task reaches a checkpoint, it checks for migration requests performed by the master daemon. If the migration is taken, they suspend their execution waiting to be deallocated and restore to another processor from the migration middleware. The actions of the master and slave daemons supporting the migration mechanism are described below.

The master daemon performs four operations:
1. The master periodically reads a data structure in shared memory where each slave daemon writes the statistics related to the processor where it runs (e.g. processor utilization and memory occupation of each task). At run-time, the master daemon processes this data and eventually issues a task migration request.
2. When a new task or an application (i.e. a set of tasks) is launched by the user the master daemon sends a message to each slave communicating that the application should be initialized. The master decides where the tasks have to be instantiated and communicates its decision to the slave daemons. The communication between master and slave daemons is implemented using dedicated, interrupt-based messages in shared memory.
3. When the master daemon wants to migrate a task, it signals to the slave daemons of the source processor that a task has to be migrated.
4. The master daemon keeps also track of the completion of applications and tasks on the various processors.

The slave daemon performs four operations:
1. It generates a new task upon notification of the master daemon. Each task is then stopped and placed in the suspended tasks queue.
2. It periodically writes in a dedicated data structure in shared memory the statistics related to tasks execution that will be used by task allocation and migration policies.
3. When the master signals that a task has to be migrated from its own processor to a target processor, it performs the following actions: i) it waits until the task to be migrated reaches a checkpoint, and puts it in the queue of the suspended tasks; ii) it copies all the task-related information (user and kernel level data structures) to a dedicated buffer in the shared memory; iii) it communicates to the slave daemon of the processor where the task must be moved that the data of the task is ready to be copied; iv) it puts the migrated task PCB in the suspended tasks queue.
4. when the slave daemon of the processor source notifies a migration request to the target daemon, the latter copies the data from the shared memory to its private memory. Finally it puts the PCB of the incoming task in the ready queue.

5. MPOS MPSOC THERMAL EMULATION FLOW

The hardware and software support we developed allows to explore resource management policies. In particular, thanks to the thermal monitoring support we implemented, it is possible to assess the impact of task migration and scheduling on system temperature as well as to design thermal aware policies. An handshake mechanism between the thermal model and the middleware had to be implemented to this purpose.

Figure 3 depicts the flow that needs to be used in order to build a custom MPOS MPSoC design. It is similar to the baseline flow described in [3] from the hardware, but it has been extended on the hardware and software side to include the MPOS support. In this case, the MB software binaries are now generated using a uClinux toolchain that enables to include OS support in the same image as the application binaries to be executed. When the designer describes an MPSoC architecture, all the information related to the hardware resources present in the system (included processing cores, additional I/O blocks, memory addresses, custom parameters, interrupts numbers...) is embedded into a configuration file that follows a special syntax, such that it can be fed into the uClinux toolchain. This toolchain subsequently allows the designer to build a custom uClinux OS image that is tailored to his particular needs from the hardware viewpoint, and where the designer can compile the applications to run in the final MPSoC. Then, using the generated configuration file, the user can easily choose the number of semaphores to use, enable/disable debugging support and thermal monitoring services, etc. Provided this information, together with the available drivers for the included hardware resources indicated in the configuration file, a complete binary file is generated, containing the compiled uClinux image to be downloaded into the MBs memories. The OS image comprehends also the file system, where the middleware and user level applications are placed after being cross-compiled, so that they are available to be started from the uClinux consoles of the microblazes after the boot.

After the configuration of the MPOS MPSoC has been done and the uClinux images have been downloaded, during the emulation the thermal model indicated in Subsection 3.1 uses the statistics collected by the sniffers included in our framework to compute power density of various chip components. In case of processing cores, their power density depends on the frequency. For this reason, the frequencies currently used are also included among the interchanged statistics.

The block diagram of the system is described in Figure1. The emulation is triggered by a timer interfaced to the PLB bus, where the PPC is connected. The timer generates and interrupt for the PPC that triggers the statistics collection. The timer can be programmed to control statistics download intervals and the period of temperature updates by writing a control word in the status register of the statistic collection peripheral.

Corresponding to each interrupt, the virtual clock manager freezes the clock of all the MB buses and stops the statistics counters.

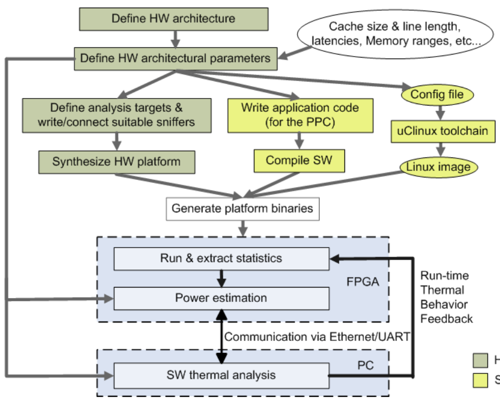
Define HW architecture
Cache size & line length, latencies, Memory ranges, etc...
Define HW architectural parameters
Config file
Define analysis targets & write/connect suitable sniffers
Write application code (for the PPC)
uClinux toolchain
Synthesize HW platform
Compile SW
Linux image
Generate platform binaries
Run & extract statistics
FPGA
Power estimation
Run-time Thermal Behavior Feedback
Communication via Ethernet/UART
SW thermal analysis
PC
HW
SW

Figure 3: Complete HW-SW flows included in the FPGA-based thermal emulation framework

The PPC reads all the statistics data and uploads them through the UART to a program in the host PC. The PPC remains idle waiting for a special character from the UART. In the meantime, the program in the host PC updates the frequencies and computes the temperatures using the thermal model. The temperatures are then sent back to the PPC through the UART. The PPC writes the temperature values into a memory mapped register. This mechanism emulates the presence of temperature sensors on the chip.

Once updated the emulated temperatures sensors, the PPC instructs the statistic peripheral to re-enable the clocks and the statistics counters. It can be noted how the whole process is completely transparent to the emulation, therefore no statistics data loss occurs and the emulation accuracy is preserved. Exploiting the emulated temperature sensors, the middleware can implement a thermal-aware task migration strategy. An example policy is described in Section 6.

6. EXPERIMENTAL RESULTS

To assess the effectiveness of the emulation framework, in this section we show experiments concerning the evolution of the temperature of an MPSoC architecture including 4 cores, when frequency scaling and task migration are available at the OS level to perform thermal management of the final chip. Each core has a 64KB cacheable private memory and 32KB of shared memory implemented both in the DDR memory. The considered floorplan is shown in Figure 4 and in our experiments the frequencies of the cores are the main monitored elements from the 128 thermal cells that can be currently considered [3]. Cores considered in the floorplan are ARM11, with frequency range of 100-512MHz, while the interconnection is an AMBA bus system. In the floorplan, processor 1 refers to MB0, processor 2 refers to MB1 and so on. In the emulation system, cores are microblazes whose frequency range is from 10-51.2MHz. As such, they are ten times slower. In the thermal model, we compute the power consumption of the bus by considering the access patterns generated by the cores [16]. We have obtained the dimensions of the AMBA circuits by synthesizing and building a layout. The dimensions of the memories and processors are based on numbers provided by an industrial partner. As software driver for this MPSoC design, we have defined a benchmark that stresses the processing power of the MPSoC design to observe effects in temperature. This benchmark implements a synthetic task that imposes a load near 100%.

In the first experiment, shown in Figure 5, we run the synthetic task on the MB1. We can observe that the temperature of MB1

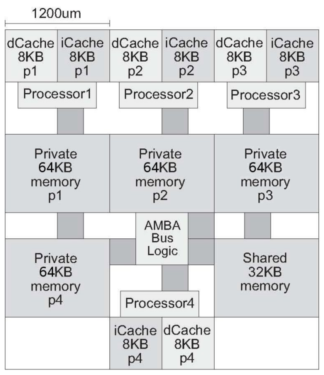

Figure 4: MPSoC floorplan with uneven distribution of cores on the die and a shared bus interconnect

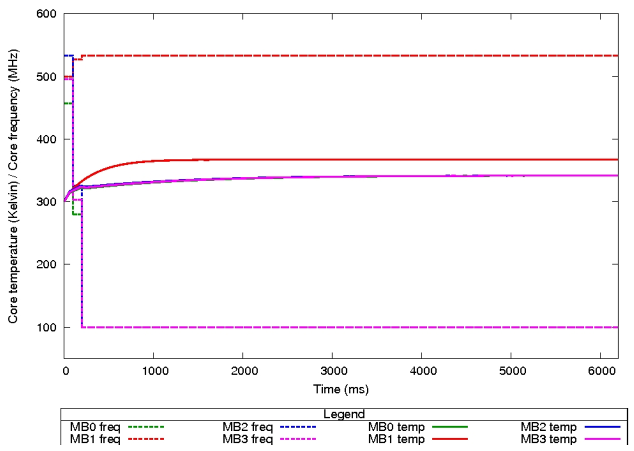

Figure 5: Temperature waveform with one task running on MB1

increases from the initial ambient temperature (300 Kelvin degrees) and stabilizes to a value that depends on the processor load. In fact, the OS in each processor automatically adjusts the frequency of the core depending on the load using a frequency setting policy based on the observation of processor load over time intervals [8]. In the thermal model, the frequency information is used to feed power models of the processor to compute its power consumption. In this experiment, the other cores run at the minimum frequency (100MHz). It must be noted how their temperatures are affected by MB1, however, being all the processors unloaded, they stay below 340K.

In this second experiment we show the effect of a thermal-aware task migration policy (Figure 6). In this case, a synthetic task is running on the MB1 which can migrate among the available cores. To this end, the middleware system periodically monitors processor temperatures and compares them with a threshold. We set this threshold to be 365 Kelvin degrees in this experiment.

The curves in Figure 6 show temperature and frequency waveforms of each core over time. It can be observed that, as the temperature of MB1 reaches the threshold, the middleware system triggers the task migration to the colder processor MB2. Therefore, the temperature of MB1 decreases while temperature of MB2 increases and reaches the threshold triggering another task migration to MB3. In this experiment we kept MB0 unloaded to observe its load-free temperature behavior, that is not affected by the tempera-

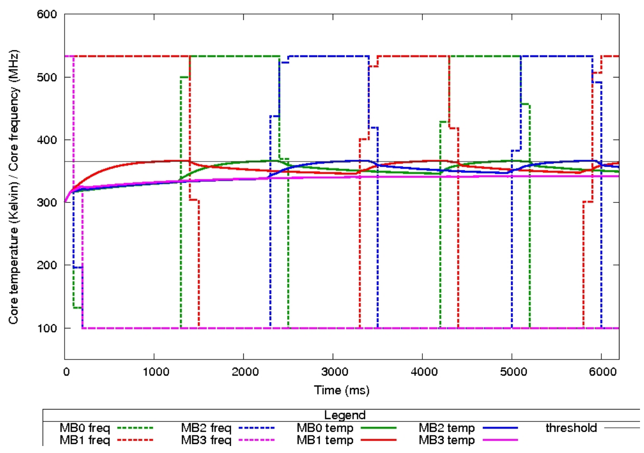

Figure 6: Temperature effect of a simple temperature-aware task migration policy.

ture of other processors in this design.

From this simple experiment we can observe several interesting consequences of MPSoC temperature management: i) the temperature of each core is affected by the others but strongly depends on the load, which can be efficiently monitored by the OS since this layer has full knowledge of task being executed and, even more importantly, which are the following tasks that need to be executed. Hence, the OS can define a proper task migration policy according to possible prior (design) knowledge of the location of the cores in the floorplan and the thermal conductivity between their cells; ii) thermal time constants are larger with respect to task migration delays; Thus, task migration can be effective in controlling cores' temperatures. However, task migration imposes an overhead due to data exchange between processors and to task shut-off and resume delays. Therefore, in principle the number of migrations per time unit is limited. Nevertheless, as our results indicate, since temperature variations are slow with respect to our implemented migration overhead, moving tasks between processors is a viable technique to keep chip temperature controlled.

Finally, regarding exploration efficiency, our results show the duration of both experiments was approximately 90 seconds for 6 seconds of real-time, which indicates more than 1000 times speed-up w.r.t. cycle-accurate MPSoC simulators including OS [16]. Emulation time depends on two contributions: i) the processing cores are ten times slower than the emulated system; ii) there is an additional time overhead to synchronize with the thermal simulation library at run-time and to download statistics to the host PC. Overall, the performance of the emulation is efficient enough for very fast system prototyping and MPOS thermal validation.

7. CONCLUSIONS

In this paper we have presented a new emulation framework that enables the rapid evaluation and exploration of MPOS implementations for MPSoC designs by using conventional FPGAs. It includes all the architectural support at the hardware level to validate different inter-processor communication and task scheduling schemes. Moreover, our framework enables long thermal emulations of MPOS implementations running onto MPSoCs architectures, and our results show the benefits of this framework to explore thermal-aware management at the OS level. In the future we would like to study more in detail the relationship of complex OS-based thermal management techniques with reliability of MPSoCs.

8. REFERENCES

[1] Aptix. System explore, 2003. `http://www.aptix.com`.

[2] ARM. Arm integrator ap, 2004. `http://www.arm.com`.

[3] D. Atienza, P. G. Del Valle, G. Paci, F. Poletti, L. Benini, G. De Micheli, and Jose M. Mendias. A fast hw/sw fpga-based thermal emulation framework for multi-processor system-on-chip. In *Proceedings of Design Automation Conference (DAC)*, pages 618–623. ACM Press, 2006.

[4] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri. Mparm: Exploring the multi-processor SoC design space with SystemC. *The Journal of VLSI Signal Processing*, 41(2):169–182, September 2005.

[5] Cadence. Cadence palladium ii, 2005. `http://www.cadence.com`.

[6] Emulation and Verification Engineering. Zebu xl and zv models, 2005. `http://www.eve-team.com`.

[7] Heron Engineering. Heron mpsoc emulation, 2004. `http://www.hunteng.co.uk`.

[8] Krisztian Flautner and Trevor Mudge. Vertigo: automatic performance-setting for linux. volume 36, pages 105–116, New York, NY, USA, 2002. ACM Press.

[9] Ahmed Jerraya and Wayne Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, Elsevier, 2005.

[10] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[11] Berkeley University RAD Lab. Ramp project, 2006. `http://radlab.cs.berkeley.edu/wiki/RAMP_project_idea`.

[12] Sergio López-Buedo, Javier Garrido, and Eduardo I. Boemo. Thermal testing on reconfigurable computers. *IEEE Design & Test of Computers*, 17(1):84–91, 2000.

[13] Y. Nakamura, K. Hosokawa, I. Kuroda, K. Yoshikawa, and T. Yoshimura. A fast hardware/software co-verification method for system-on-a-chip by using a c/c++ simulator and fpga emulator with shared register communication. In *Proceedings of Design Automation Conference (DAC)*, pages 299–304, 2004.

[14] M. Diaz Nava and et al. An open platform for developing mpsocs. *IEEE Computer*, pages 60–67, 2005.

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

[16] G. Paci, P. Marchal, F. Poletti, and L. Benini. Exploring "temperature-aware" design in low-power mpsocs. In *Proceedings of the conference on Design, automation and test in Europe (DATE)*, pages 838–843. IEEE/ACM, 2006.

[17] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *Transaction on Architectures and Code Optimizations (TACO)*, 1(1):94–125, 2004.

[18] H. Su, F. Liu, A. Devgan., E. Acar, and S. Nassif. Full chip leakage estimation considering power supply and temperature variations. In *Proc. IEEE/ACM ISLPED*, pages 78–83, Aug. 2003.

[19] O. Takahashi, S. R. Cottier, S. H. Dhong, B. K. Flachs, and J. Silberman. Power-conscious design of the cell processor's synergistic processor element. *IEEE Micro*, 25(5):10–18, 2005.