Evaluating the Impact of Task Migration in Multi-Processor Systems-on-Chip

Gabriel Marchesan Almeida, Sameer Varyani, Rémi Busseuil,
Gilles Sassatelli, Pascal Benoit, Lionel Torres
Laboratory of Informatics, Robotics and Microelectronics of Montpellier (LIRMM)
Department of Microelectronics
161 Rue Ada, Cedex 5, 34095
Montpellier, France
{marchesan, varyani, busseuil, sassatelli, benoit, torres}@lirmm.fr

Everton Alceu Carara, Fernando Gehm Moraes
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Faculty of Informatics
Av. Ipiranga, 6681 - Partenon - CEP: 90619-900
Porto Alegre, RS, Brazil
{carara, moraes}@pucrs.br

ABSTRACT

This paper presents a Multi-Processor System-on-Chip platform which is capable of load balancing at run-time. The system is purely distributed in the sense that each processor is capable of making decisions on its own, without having relying by any central unit. All the management is ensured by a very tiny preemptive RTOS (run-time operating system) running on every processor which is mainly responsible for running and distributing tasks among the processing elements (PEs). The goal of such strategy is to improve the performance of the system while ensuring scalability of the design. In order to validate the concepts, we have conducted some experiments with a widely used multimedia application: the MJPEG (Motion JPEG) decoder. Obtained results show that the overhead caused by the task migration mechanism is amortized by the gain in term of performance.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real- time and embedded systems; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network operating system*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Interconnection architectures*

General Terms

Design, Experimentation, Performance

Keywords

MPSoC, task migration, homogeneous, adaptive, NoC, distributed memory, RTOS

1. INTRODUCTION

Multi-Processor Systems-on-Chip (MPSoC) are attractive candidate architectures for multimedia processing as multimedia schemes generally can be partitioned in data-dominated functions, which can be processed in parallel on different cores. MPSoCs may be classified into two different categories: heterogeneous and homogeneous architectures. Heterogeneous architectures achieve a better performance/power consumption trade-off at the cost of reduced flexibility. Homogeneous architectures on their side open interesting perspective in the area of on-line adaptation such as workload balancing [1] or fault tolerance thanks to their intrinsic functional redundancy.

Since this work targets massively parallel on-chip multiprocessor systems, scalability is a major concern in the approach. In the direction of scalable systems we have developed a homogeneous MPSoC architecture with distributed memory making use of a message passing programming model.

Given the large variety of possible use cases that these platforms must support and the resulting workload variability, offline approaches are no longer sufficient as application mapping paradigms, because they do not allow coping with time changing workloads. Furthermore, the large number of parameters that play a role on the platform performance makes it difficult to estimate the best system response at design time.

In [2] we have proposed an adaptive strategy which is responsible for making decisions at run-time. Decisions are taken by processors in a distributed fashion and relate mostly to application performance. This present paper puts focus on the task migration strategy, and mostly addresses both the benefits brought by this technique as well as the associated performance penalty. The new architecture relies on a

RTOS with support for semaphores, mutexes and task priority based scheduling algorithm. Moreover, a tiny implementation of a communication stack comprising UDP TCP/IP protocols is available.

The remainder of the paper is organized as follows: Section 2 presents some of the existing software/hardware architectures that enable task migration in the context of MPSoC. Section 3 explains the task migration support which is the basis of this work. In Section 4 experimental results are presented while conclusions are drawn in Section 5.

2. RELATED WORKS

In recent years due to the growing interest for MPSoCs, task migration mechanisms have been explored in order to deal with problems related to load balancing among PEs. Some works have adopted the benefits of such strategies for thermal balancing [3][4][5].

2.1 Task migration using shared memory

For shared memory systems such as today's multicore computers, task migration is facilitated by the fact that no data or code has to be moved across physical memories: since all processors are entitled to access any location in the shared memory, migrating a task comes down to electing a different processor for execution. There exist several efficient implementations on general purposes OS such as Windows or Linux.

Task migration has been explored for MPSoCs, notably based on locality considerations [6] for decreasing communication overhead or power consumption. In [7], authors present a migration case study for MPSoCs that relies on the μClinux operating system and a check pointing mechanism. The system uses the MPARM framework [8], and although several memories are used, the whole system supports data coherency through a shared memory view of the system.

In [9] authors propose to implement a scalable shared memory many-cores architecture with global cache coherence. The architecture is built around 4096 cores which makes use of a logically shared memory but physically distributed, with cache coherence enforced by hardware, using a directory-based protocol.

2.2 Task migration using distributed memory

In the case of distributed memory MPSoCs, both process code and state have to be migrated from a processor private memory to another, and synchronizations must be performed using exchanged messages. While this proves straightforward in typical general purpose computers thanks to the presence of a memory management unit (MMU), implementing task migration on tiny MMU-less embedded processors is challenging.

In [10] authors propose a policy which exploits run-time temperature as well as workload information of streaming applications to define suitable run-time thermal migration patterns. In their approach each processor relies on having a replica of the task in memory. When the migration occurs, the processor stops the execution of the task and the processor in the destination starts running its own replica. Their architecture is based on distributed memory and the processors explicitly communicate with each others by means of the shared memory.

This mechanism works well but is very limited in terms of memory overhead once every processor has to have a copy of the task in its memory.

The question which may arise when using shared memory for enabling task migration is concerning the scalability of the system. There is a strong tendency for the next generation of homogeneous MPSoC in using systems with distributed memory targeting scalable and massively parallel architectures.

A number of work in the literature based on distributed memory systems has been using shared memory as a mean for enabling task migration [7][11][12]. In [11] each core runs a single operating system instance in its logical private memory. Processor cores execute tasks from their private memory and explicitly communicate with each others by means of shared memory. The target platform uses a shared bus as interconnect.

In [13] a dynamic task allocation strategy is proposed. The work evaluates task allocation strategies based on bin-packing algorithms in the context of MPSoCs. The mechanism adopted is based on a copy model. The whole context (code, data, stack, and contents of internal registers) is migrated and there is no task execution during the transfer. The interprocessor communication is based on send/receive primitives. However, in this work neither explanation about the task migration protocol nor the impact in term of performance of such mechanism is given.

Taking into account the future homogeneous MPSoC systems tendency, scalable architectures with purely distributed memory system are suitable. To the best of our knowledge, our architecture is the only one with a purely distributed memory system in which does not rely on using shared memory for enabling task migration. Instead, it uses the NoC as a communication link where tasks are transmitted during the migration process.

3. TASK MIGRATION SUPPORT

To handle dynamic conditions where processors may become overloaded very rapidly, we implemented a task migration mechanism which enables this feature. Migration policies can exploit this mechanism to perform load balancing for achieving better performance or for saving energy.

3.1 Architectural description

The architecture is made of a homogeneous array of Processing Elements (PEs) communicating through a packet-switching network. For this reason, the PE is called NPU (Network Processing Unit). Each NPU has multitasking capabilities which enable time-sliced execution of multiple tasks. This is supported by a tiny preemptive multitasking Operating System which runs on each NPU. This system further provides usual services such as queues, threads, semaphores and mutexes. The entire operating system has a memory footprint of $55KB$, which comprises the dynamic task loader we developed to the purpose of enabling task migration strategies described in this paper. The structural view of the platform is depicted in Figure 1.

The NPU is built of two main layers, the network layer and the processing layer. The network layer is essentially a compact routing engine based on the Hamiltonian Routing Algorithm. The Network-on-Chip (NoC) used in this work was proposed in [14].

The purpose of this paper being to study and analyze

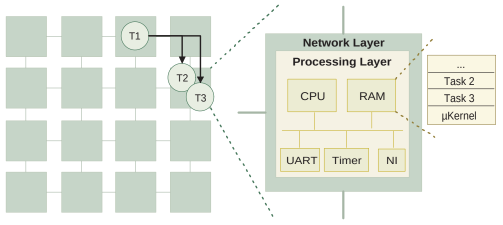

Figure 1: Structural View of the Platform

the performance penalty induced by task migration and not inter-processor communication, we use in all experiments a NoC which features a bandwidth largely over dimensioned with respect to the processor performance. In the validation scenarios we use a 2×2 mesh-topology network.

Packets are read from incoming physical ports, then forwarded to either outgoing ports or the processing layer. Whenever a packet header specifies the current NPU address, the packet is forwarded to the network interface (NI in Figure 1). The network interface buffers incoming data in a small hardware FIFO and simultaneously triggers an interrupt to the processing layer. The interrupt then activates data demultiplexing from the single hardware FIFO to the appropriate software FIFO as illustrated in Figure 2.

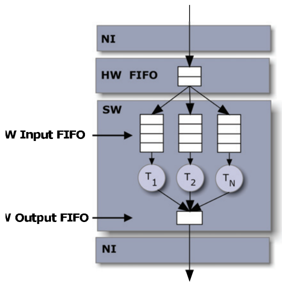

Figure 2: Functional View of a NPU

The processing layer is based on a simple and compact RISC microprocessor, its static memory, and a few peripherals (one timer, one interrupt controller, one UART) as shown in Figure 1. The processor used has a compact instruction set comparable to a MIPS-1. It has 3 pipelines stages, no cache, no Memory Management Unit (MMU), and no memory protection support in order to keep it as small as possible. A multitasking tiny Real-Time Operating System (RTOS) implements the support for time-multiplexed execution of multiple tasks. The original version of the RTOS as well the RTL description of the processor used as part of this work are available at [15].

3.2 Memory management strategy

As described in Section 2.2 implementing task migration mechanisms in architectures without MMU is a complex task. To provide task migration support in such architectures without having to rely on using task replication approach, the system has to be capable of loading task dynamically while ensuring memory protection in order to prevent task access memory that has not been allocated to it. Basically there are two main ways to define the system memory space for storing tasks.

3.2.1 Paging

The first one is called paging and consists on dividing the memory into equal, small pieces, called pages. Virtual memory makes it possible to have a linear virtual memory address space and to use it to access blocks fragmented over physical memory address space. The first problem when using this approach is the fact that for every application it is necessary to tune the page size in order to be big enough for storing tasks. Due the fact the applications may vary a lot in terms of size, this approach becomes not scalable since a huge amount of memory can be wasted.

3.2.2 Dynamic memory allocation

The second way is to use mechanisms that allow to dynamically allocate/deallocate memory. This approach is more convenient when scalability is targeted. Therefore it is more difficult to manage once the RTOS has to resolve memory address references. By means of using a mechanism which has a linked list of occupied memory address the RTOS can avoid memory conflicts. The problem by using such approach is the fact that as soon as we have tasks migrating very often, the system memory becomes fragmented. Therefore this approach still looks more flexible and scalable when comparing to the paging mechanism.

Since we target many-cores architectures and then scalable systems, we have chosen the second approach. As the MMU is not present in the system we had to use a feature supported by the GCC compiler that enables to emit relocatable code named Position Independent Code (PIC). This feature is generally used for shared libraries and generates only relative jumps and accesses data locations and functions using a Global Offset Table (GOT).

3.3 Task migration protocol

The task migration protocol is depicted in Figure 3. In this scenario the considered application is split into 3 tasks (T_1, T_2, T_3). At the beginning of the processing tasks are running in different NPUs$_{(0\ldots3)}$. At a given moment NPU$_1$ decides migrating T_2 to NPU$_0$ (Step $_1$). The NPU$_1$ sends a control packet to the master of the cluster (NPU$_1$) asking authorization for performing task migration (Step $_2$). It is important to highlight that NPU$_0$ is master in the sense that there is a global routing table[1] in this node, but each NPU takes its own decisions, characterizing a purely distributed system. The master of the cluster checks in its routing table if there is one or more tasks sending data to the task which is supposed to be migrated. In this case, T_1 and T_3. Then, the master sends a control packet to these tasks asking the tasks to stop sending packet to T_2 so that migration can be initiated. In this paper we have performed all the experiments in a single cluster built of 4 nodes. For future experiments where larger MPSoCs architectures will be used the model based on multi-clusters will be adopted.

Immediately after receiving the control packet from the Master, T_1 and T_3 stop sending data to T_2 (Step $_4$). Then, T_2 reads out the content of its FIFOs and then is migrated to NPU$_0$ (Step $_5$). During the migration process only the opcode of the task is sent over the NoC. Further, as the processor is not equipped with a MMU that handles virtual to physical address translation, migrating task context is not supported in the system. Therefore application tasks that

[1]Data structure which stores task locations

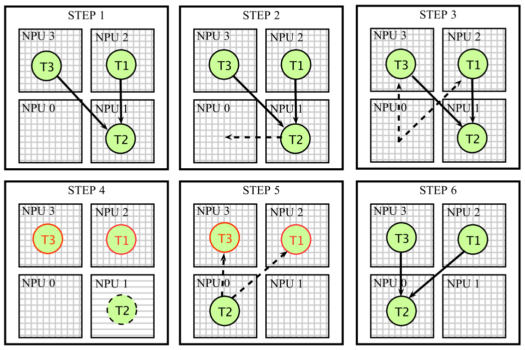

Figure 3: Overview of the Task Migration Protocol

maintain parameters across processed blocks (such as adaptive filters) cannot be migrated. This is rarely limitative as most streaming applications do not require this. The next step is registering the task into the memory and inserting it into the scheduling list. This process is done as soon as the task reaches the destination. Then, in (Step $_6$) the master node sends a control packet to T_1 and T_3 informing that T_2 has been migrated and then the data transfer can be resumed. This control packet carries both the message for resuming sender tasks and new task location.

Note that compared to other approaches [12], this protocol implements a non-packet-loss control. The packets transmitted during the migration process will be received and processed.

4. EXPERIMENTAL RESULTS

In this section we evaluate the overhead induced by the proposed task migration mechanism. In order to fairly assess the performance overhead induced by the proposed task migration strategy, experiments are conducted in best effort mode, therefore all implementations aim at maximizing performance rather than ensuring quality of service. This allows stressing the architecture as much as possible, hence migration cost become prominent rather than compensated by transient increase in CPU usage.

For performing the experiments we have used ModelSim as SystemC/VHDL simulator. The entire architecture is described in both SystemC and VHDL RTL, except the NoC which is not available in SystemC. The MIPS processor is coded based on an ISS (Instruction Set Simulator), built around of a SystemC wrapper which connects the processor to the rest of the components in the architecture.

Multimedia applications are characterized by soft real-time requirements. As a consequence, migration overheads must be carefully evaluated to prevent deadline misses when moving processes between nodes [11]. In order to validate our task migration mechanism and perform our experiments, we have implemented a MJPEG (Motion JPEG) decoder.

4.1 MJPEG profiling and partitioning

In order to better measure the workload of each function in the MJPEG application, we have performed an application profiling. It is possible to clearly identify the three main functions in the MJPEG decoder (IVLC, IDCT and IQUANT) as well the most critical (IVLC). Based on this information, we have partitioned the MJPEG decoder by

Table 1: Static mappings for the MJPEG decoder

Mapping		NPU			
		0	1	2	3
SM_1		T_2, T_3, T_4	-	-	T_1
SM_2		T_3, T_4	T_2	-	T_1
SM_3	Tasks	T_2	T_3	T_4	T_1
SM_4		T_2, T_4	T_3	-	T_1
SM_5		T_2, T_3	T_4	-	T_1

representing it as a task graph built of four tasks ($T_1 \rightarrow$ Sender, $T_2 \rightarrow$ IVLC, $T_3 \rightarrow$ IQUANT and $T_4 \rightarrow$ IDCT).

4.2 MJPEG decoder statically mapped

Aiming to compare static mappings against dynamic mapping mechanisms we have created five different scenarios which use static mapping ($SM_{1...5}$). Table 1 presents the placement for the tasks. For all the scenarios we have considered putting T_1 in NPU_3. This task is not part of the MJPEG decoder itself and it only responsible for feeding the decoder with the input streams. In the first mapping (SM_1) all the three tasks are running in the NPU_0, in the second mapping (SM_2) the most critical task (IVLC) is running alone in NPU_1, etc.

Figure 4 shows the performance of the MJPEG decoder when mapped to different NPUs. It is important to observe that in the worst case (SM_5) the communication cost plays an important role in the performance of the system. In the scenario which achieves the best performance (SM_2), the most critical task is mapped into a dedicated NPU.

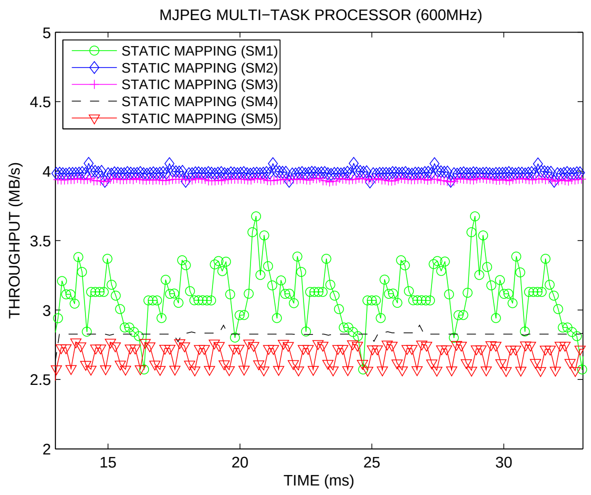

Figure 4: Different Static Mappings

It is very clear that whenever critical tasks are mapped into dedicated NPUs performance is improved, the problem is to evaluate the performance for every possible mapping when the number of tasks and the complexity increases. The number of possible mappings is given by N^T, where N represents the number of NPUs and T the number of tasks. So, for a system with 4 processors and 20 tasks, the number of possible combinations would be equal to $1.20e+24$, which makes the off-line mapping estimation unfeasible.

4.3 MJPEG decoder dynamically mapped

In order to facilitate the comprehension of our task migration mechanism we have defined some terms:

- M_T: The task candidate to be migrated;
- $P_{i,j}$: List of predecessors tasks of task j;
- $S_{i,j}$: List of successors tasks of task j;
- MT_j: Timestamp corresponding to task migration mechanism triggered for task j;
- MS_j: Timestamp corresponding to task j migration start;
- MF_j: Timestamp corresponding to task j arrival at destination NPU, comprising operating system task registration in the ready list;

Aiming to deal with such huge amount of possibilities, the system has to be capable of adapting at run-time. The basic mechanism for enabling adaptability in a purely distributed MPSoC system is through the task migration. Aiming to evaluate the overhead caused by the system when using such mechanism we have created a scenario as follows:

At the beginning of the processing all three tasks of the MJPEG application are mapped to NPU$_0$. Additional to them, there is a task responsible for feeding the system with the stream to be decoded. Then the most critical task (IVLC) is migrated to NPU$_1$. The selection of the NPU where the task will be migrated is beyond the scope of this paper. The task migration mechanism is triggered from a thread defined in the RTOS. Figure 5 shows the throughput during the task migration process.

At the moment the task migration process is triggered, the master node asks all predecessor tasks ($P_{i,2}$) of M_2 to stop sending packets. At this moment the task which feeds information to the system stops sending packets to M_2.

At $MS_2 = 16ms$ NPU$_0$ starts transmitting T_2 over the NoC. From this moment up to MF_2, we clearly see the performance decreases. Once received and registered in memory (MF_2), T_2 is ready to be scheduled. Our task migration protocol implements a non-packet-loss control, before migrating a given task all the data in its incoming FIFO are read out and processed. After task has been migrated, $P_{i,2}$ resumes sending packets to T_2.

After the FIFO of T_2 has been fed up the performance gets increased going from 0.9MB/s to around 4MB/s. In average the gain in term of performance it is about 25% when compared to the initial static mapping (3MB/s).

To the purpose of assessing the penalty induced by the migration in the used best-effort implementations we conducted experiments for finding the break-event point.

Figure 5 shows a plot in which we consider a buffer which is fed by the MJPEG processing pipeline output data. Data are read out from that buffer at a rate corresponding to 3MB/s, aligned to the first performance plateau.

During this phase, buffer filling is therefore constant as data are written and read out at the same rate. At $MT_2 = 7.5ms$ migration decision is issued, which translates in an immediate drop in throughput and therefore buffer filling that extends until the end of the migration process. As soon as the migration is completed throughput rapidly reaches a higher plateau that in turn translates in a rapid increase in buffer filling. The initial buffer level is reached at time $42ms$ and at that time the penalty incurred by the migration

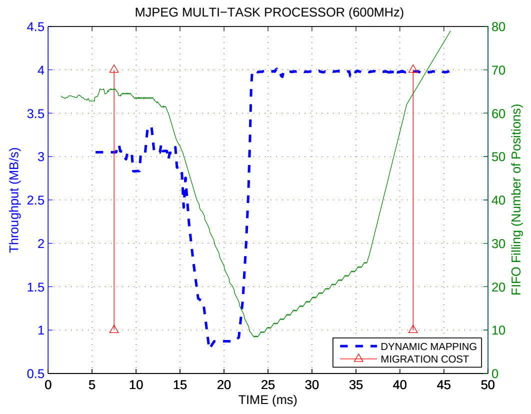

Figure 5: Throughput During Task Migration

is fully compensated. The break-even point is therefore in the order of $34.5ms$ in this scenario. Figure 6 presents the timing jitter for the dynamic mapping.

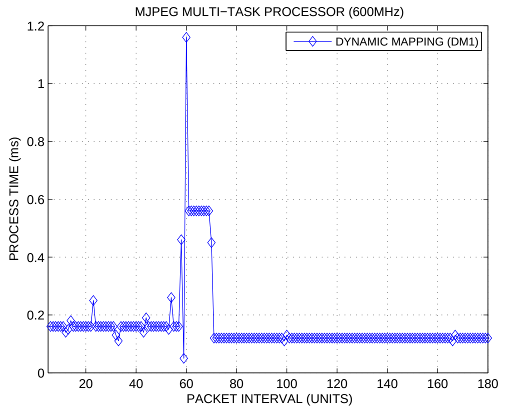

Figure 6: Timing Jitter (Dynamic Mapping)

We can observe that at the beginning of the processing, before triggering the task migration, the timing jitter oscillates. The reason is that since all the tasks share the same NPU there is no pipeline and then the packet arrival is not constant. During the task migration process the jitter is increased due to the fact that the execution flow is stopped. At the moment of the task migration we can clearly observe the timing jitter decreasing and then getting stable. This is explained by the fact that the new mapping provides a better task mapping.

Figure 7 presents three different mappings for the MJPEG application previously used. There are two static mappings without task migration and one dynamic mapping with task migration. For the two static mappings we have used two with best/worst performance respectively defined as SM$_2$ and SM$_5$. For the dynamic mapping the initial mapping place all three MJPEG tasks in NPU$_0$ and T_1 in NPU$_3$ and later the most critical task is migrated to NPU$_1$ (SM$_2$).

Obtained results show that even under heavy load the proposed multiprocessor system is capable of completing a task migration in a tenth of milliseconds, and that the penalty is compensated in a few tenths of milliseconds. This result sug-

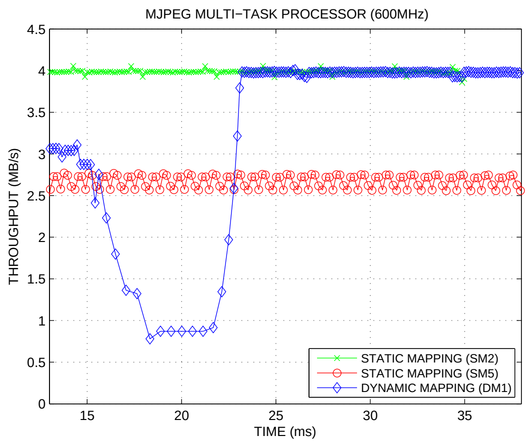

Figure 7: Static vs Dynamic Mapping

gests that dynamic load balancing with frequent task remapping may provide significant benefits over traditional design-time decided mappings. Further, the observed performance overhead resulted from best-effort application implementations. In the case of soft or hard real-time implementations where processors are sized to purpose and therefore rarely overused, adequate buffering may completely hide performance cost therefore resulting in virtually negating task migration overheads.

5. CONCLUSION

In this paper we have evaluated the impact caused by the task migration mechanism here proposed. We analyze the efficiency of our approach when compared to traditional approaches. The results show that the overhead produced by the mechanism is amortized by the gain in term of performance which is around 25% when compared to static mapping for the case of the MJPEG application.

For validating the experiments we have used a 2×2 homogeneous MPSoC architecture. It implements a task migration mechanism in order to provide load balancing capabilities. We have compared our approach against the existing ones and to the best of our knowledge, our architecture is the only one with a purely distributed memory system that does not rely on hared memory for enabling task migration.

As future work we plan to implement more complex scenarios with synthetic tasks for better exploiting/tuning all the parameters aiming to have an efficient task migration mechanism for MPSoC. Also we aim to use task migration mechanism in order to deal with power consumption issues once by using such strategy we can better harmonize the system and save power consumption.

6. REFERENCES

[1] Rainer L., Andras V., Marco B., Soonhoi H., Rainer D., and Achim N. Programming mpsoc platforms: Road works ahead! In *DATE'2009* [1], pages 1584–1589.

[2] G. Marchesan Almeida, G. Sassatelli, P. Benoit, N. Saint-Jean, S. Varyani, L. Torres, and M. Robert. An adaptive message passing mpsoc framework. *International Journal of Reconfigurable Computing*, Volume October, 2009.

[3] P. Chaparro, J. Gonzalez, G. Magklis, Cai Qiong, and A. Gonzalez. Understanding the thermal implications of multi-core architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 18(8):1055–1065, August 2007.

[4] T. Sato, J. Ichimiya, N. Ono, K. Hachiya, and M. Hashimoto. On-chip thermal gradient analysis and temperature flattening for soc design. In *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 1074–1077, New York, NY, USA, 2005. ACM.

[5] I. Yeo, C. C. Liu, and E. J. Kim. Predictive dynamic thermal management for multicore systems. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 734–739, New York, NY, USA, 2008. ACM.

[6] A. Barak, O. La'adan, and A. Shiloh. Scalable cluster computing with mosix for linux. In *In Proceedings of Linux Expo'99*, pages 95–100, 1999.

[7] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali. Supporting task migration in multi-processor systems-on-chip: A feasibility study. In *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, volume 1, pages 1–6, 2006.

[8] D. Bertozzi L. Benini. Mparm: Exploring the multi-processor soc design space with systemc. *The Journal of VLSI Signal Processing*, 41(2):169–182, 2005.

[9] Alain Greiner. Tsar: a scalable, shared memory, many-cores architecture with global cache coherence. In *9th International Forum on Embedded MPSoC and Multicore (MPSoC'09)*, Savannah, Georgia, USA, 2009. IEEE Press.

[10] F. Mulas, D. Atienza, A. Acquaviva, S. Carta, L. Benini, and G. De Micheli. Thermal balancing policy for multiprocessor stream computing platforms. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(12):1870–1882, 2009.

[11] A. Acquaviva, A. Alimonda, S. Carta, and M. Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP J. Embedded Syst.*, 2008:1–15, 2008.

[12] M. Pittau, A. Alimonda, S. Carta, and A. Acquaviva. Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation. In Samarjit Chakraborty and Petru Eles, editors, *ESTImedia*, pages 59–64. IEEE, 2007.

[13] Daniel Barcelos, Eduardo Wenzel Briao, and Flávio Rech Wagner. A hybrid memory organization to enhance task migration and dynamic task allocation in noc-based mpsocs. In *SBCCI '07:*, pages 282–287, New York, NY, USA, 2007. ACM.

[14] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost. Hermes: an infrastructure for low area overhead packet-switching networks on chip. *Integration, the VLSI Journal*, 38(1):69–93, 2004.

[15] S. Rhoads. Plasma - most mips i(tm) (http://www.opencores.org/project,plasma).