MPI-Based Adaptive Task Migration Support on the HS-Scale System

Nicolas Saint-Jean, Pascal Benoit, Gilles Sassatelli, Lionel Torres, Michel Robert

▶ **To cite this version:**

Nicolas Saint-Jean, Pascal Benoit, Gilles Sassatelli, Lionel Torres, Michel Robert. MPI-Based Adaptive Task Migration Support on the HS-Scale System. ISVLSI 2008 - IEEE Computer Society Annual Symposium on VLSI, Apr 2008, Montpellier, France. pp.105-110, 10.1109/ISVLSI.2008.87 . lirmm-00280687

HAL Id: lirmm-00280687

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00280687

Submitted on 9 Sep 2022

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

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

MPI-Based Adaptive Task Migration Support on the HS-Scale System

Saint-Jean N., Benoit P., Sassatelli G., Torres L., Robert M.
University of Montpellier II, LIRMM, UMR 5506
<name>@lirmm.fr

Abstract

Scalability of architecture, programming model and task control management will be a major challenge for future VLSI systems. In this context, homogeneous MPSOC is a seducing approach as it is intrinsically scalable. HS-Scale is a contribution in this domain and was already published in [1,2]. In this article, we present an original MPI-based adaptive task migration support for the HS-Scale system. Our previous communication API was modified in order to be MPI compliant. In order to enable task migration without any MMU, a Position Independent Code compilation technique is implemented. The self-adaptability is based on monitoring information collected at run-time by each processing element (PE). Each PE is endowed with the same decisional capability insuring the scalability of the solution. A MJPEG case study validated on a multi-FPGA prototyping platform is presented. The observation of the dynamic behavior of HS-Scale shows that the system is able to find itself a stable task placement providing the best performance in terms of processing throughput.

1. Introduction

Multi-Processor System-on-Chips (MPSoCs) are becoming an increasingly popular solution that combines flexibility of software along with potentially significant speedups. These complex systems usually integrate a few mid-range microprocessors for which an application is usually statically mapped at design-time. Those applications however tend to increase in complexity and often exhibit time-changing workload which makes mapping decisions sub-optimal in a number of scenarios.

Our previous work, presented in [1] and [2] aimed at exploring and defining principles which granted both hardware and software scalability, namely HS-Scale. The hardware architecture, H-Scale, is a homogeneous MP-SOC based on RISC processors, distributed memories and an asynchronous network on chip. S-Scale is a programming model handled at run-time by a compact Operating System which permits essentially to schedule tasks and to manage the memory and communications between tasks. Several experiments were conducted on HS-Scale with the implementation of several applications: FIR, DES and MJPEG. The results showed the importance of the task placement on the architecture with several task characteristics in terms of regularity and granularity. One very important result was the correlations observed between performance and task distribution allowing us to forecast some adaptive strategies to automate the task management and distribute it over the system.

In this paper, our new contribution is the complete implementation of an adaptive migration support based on the Message Passing Interface (MPI) programming model. The HS-Scale system is now based on a set of adaptive principles which endow the architecture with some decisional capabilities. Based on a distributed monitoring scheme, each processing element is able to migrate automatically its tasks following a customizable policy. These mechanisms were developed in a fully decentralized fashion in order to satisfy the scalability of our solution. Also, an originality of our contribution is that the task migration is performed without any MMU (Memory Management Unit), but enabled thanks to PIC (Processor Independent Code) compilation.

This paper is organized as follows. Section 2 presents the related work in the field of task migration techniques for MPSOC systems. Section 3 resumes our previous work on the hardware architecture H-Scale. In section 4, the operating system and the programming model based on MPI are presented. . Our migration support is then exposed in section 5; the dynamic task loading based on PIC compilation is especially detailed and run-time adaptive mechanisms are then described. In section 6, we present the basis of our experiments, a multi-board prototyping platform. We present and discuss results obtained on a case study, MJPEG.

2. Related works

Our new contribution is the implementation of a full adaptive task migration support based on the MPI programming model. Task migration has been studied in the literature in both shared and distributed memory systems over the past as it is shown in the following paragraphs.

For shared memory systems such as today regular multi-core PCs, the process is facilitated by the fact that no data has to be moved across several physical memories; there exist several efficient implementations on general purposes OS such as Windows or Linux [3]. Task migration has also been explored for MPSoCs, notably based on locality considerations [4] for decreasing communication overhead or power consumption [5]. In [6], authors present a migration case study for MPSoCs that relies on the µClinux operating system and a checkpointing mechanism. The system uses the MPARM framework [7], and although several memories are used the whole system supports data coherency through a shared memory view of the system.

Migration on message-passing systems is generally a more difficult problem since both process code and state has to be moved from a processor to another, and synchronizations must

be performed using exchanged messages such as in [8] which is a solution out of the scope of this work since it target linux computer clusters.

Some other approaches aimed at augmenting MPI for providing a support for process migration, such as [9], [10] and [11]. All these approaches target computer clusters with the typical resources of general purpose computers and are therefore hardly applicable to MPSoCs. In [12] users present similar features based on a JAVA MPI framework that provides hardware independence; they show that despite migrating tasks implies overhead which are in the order of seconds, significant speedups can be achieved.

Finally, to the best of our knowledge, no other work combines the use of a message-passing programming model, on-chip multiprocessor system and adaptive task migration which is the main goal of the presented work.

3. H-Scale architecture

The basis of our approach stands on scalable hardware architecture called H-Scale and was presented in details in [1] and [2]. The processing element, namely the "NPU" (Network Processing Unit), is first presented. The communication infrastructure is then briefly described.

3.1. Network processing Unit

The architecture is made of a homogeneous array of PE communicating through a packet-switching network. For this reason, the PE is called NPU, for Network Processing Unit. Each NPU, as detailed later, has multitasking capabilities which enable time-sliced execution of multiple tasks. This is implemented thanks to a tiny preemptive multitasking Operating System which runs on each NPU. The structural and functional views of the NPU are depicted in Figure 1.

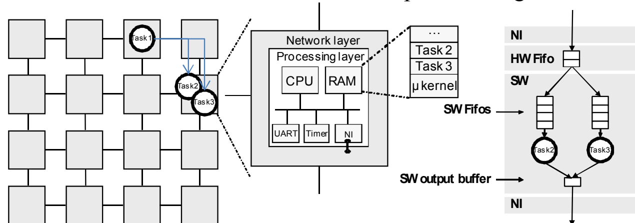

Figure 1 Network Processing Unit structural and functional descriptions

The NPU is built around two main layers, the network layer and the processing layer. The Network layer is essentially a small routing engine (XY routing). Packets are read from incoming ports, then either forwarded to outgoing ports or passed to 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 triggers data demultiplexing from the single hardware FIFO to the appropriate software FIFO as exemplified in Figure 1.

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. A multitasking microkernel implements the support for time-multiplexed execution of multiple tasks.

The microprocessor we use has a compact instruction set comparable to a MIPS-1 [13]. It has 3 pipelines stages, no cache, no Memory Management Unit (MMU) no memory protection support in order to keep it as small as possible.

3.2. Communication infrastructure

For technology-related concerns, a regular arrangement of processing elements (PEs) with only neighboring connections is favored. This helps in a) preventing using any long lines and their associated undesirable cross-talk effects in deep sub-micron CMOS technologies b) synthesizing the clock distribution network since an asynchronous communication protocol between the PEs might be used. Also, from a communication point of view, the total aggregated bandwidth of the architecture should increase proportionally with the numbers of PEs it possesses, which is granted by the principle of abstracting the communications through routing data in space. The Network-on-Chip paradigm (NoC) enables that easily thanks to packet switching and adaptive routing.

The communication framework of HS-scale is derived from the Hermes Network-on-chip, refer to [14] for more details. The routing is of wormhole type, which means that a packet is made of an arbitrary number of flits which all follow the route taken by the first one which specifies the destination address. Figure 2 depicts the simple packet format used by the network framework constituted by the array of processing elements. Incoming flits are buffered in input buffers (one per port). Arbitration follows a round-robin policy giving alternatively priority to input ports. Once access to an output port is granted, the input buffer sends the buffered flits until the entire packet is transmitted (wormhole routing).

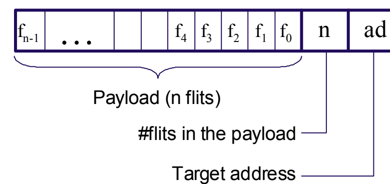

Figure 2 Packet format

Inter-NPU communications are fully asynchronous and are based on the toggle-protocol. This protocol uses two toggle signals for the synchronization, a given data being considered valid when a toggle is detected. When the data is latched, another toggle is sent back to the sender to notify the acceptance. This solution allows using completely unrelated clocks on each PE in the architecture.

4. S-scale architecture

The very first motivation for the HS-Scale system was to provide a complete scalable solution, from both hardware and software point of view. In this section, we present the

developed operating system (micro kernel) running on each NPU and insuring the scalability of our approach. Then, we present the programming model which is now MPI compliant, allowing a greater portability of the developed applications, and thus potentially higher flexibility and performance.

4.1. Distributed operating system

The lightweight operating system we use was designed for our specific needs. Despite being small (28 KB), this kernel does preemptive switching between tasks and also provides them with a set of communication primitives that are presented later. Figure 3 gives an overview of the operating system infrastructure and the services it provides.

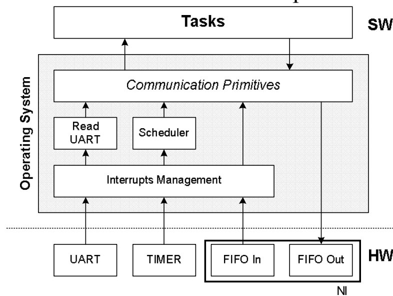

Figure 3 Operating System overview

The interrupts manager may receive interrupts from 3 hardware sources: UART, Timer and network interface (NI). Whenever an interrupt occurs, other interrupts are disabled and the processor context is saved in the system stack. Following the type of interrupt, it either reads from the UART, schedules another task (timer), receive information from other NPUs or use a communication primitive (interrupt from the network interface FIFO_in). Afterwards processor context is restored and interrupts are re-enabled. The scheduler is the core of the microkernel but is quite simple. Each time a timer interrupt occurs, it checks if there is a new task to run. In the positive case, it executes this new task. Otherwise, it has two possibilities: either there is no task to schedule then it just runs an idle task, or there is at least one task to schedule. Tasks are scheduled periodically following a round robin policy as depicted in Figure 4.

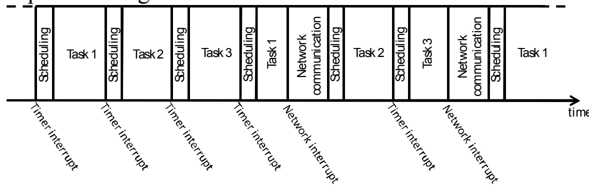

Figure 4 Scheduling diagram

4.2. MPI-Based programming model

Previously, we had developed a proprietary communication API. The first new contribution of this paper is that our API is now MPI compliant. The Message-Passing Interface (MPI) is a popular standard that is used in High Performance Computing computer clusters for instance.

Since this work targets massively parallel on-chip Multiprocessor systems scalability is a major concern in the approach. For this reason, we put focus on distributed memory machines and therefore message passing model for it provides a natural mapping to such machines. MPI being the de-facto standard, several works aimed at porting MPI subsets for MPSoCs.

Hence, tasks are hosted on NPUs which provide through their operating system communication primitives that enable data exchanges between communicating tasks. The proposed model reuses only two of the MPI communication primitives, MPI_send() and MPI_recv(). Figure 5 depicts the layered view of the communication protocol we use. MPI_recv() blocks the task until the data is available while MPI_send() is blocking until the buffer is available on the sender side. In our implementation each call exhibits this behavior and is translated into a sequence of low-level Send_Data() / Receive_Data() methods that set up a communication channel trough a simple request/acknowledge protocol. This protocol ensures the remote processor buffer has sufficient space before sending the message which help lowering the contentions in the communication network and also prevents deadlocks in it.

Figure 5 depicts the communication stack that is used in our system. Although a hardware implementation could certainly help improving performance, for compactness reasons it is fully implemented in software down to packet assembling.

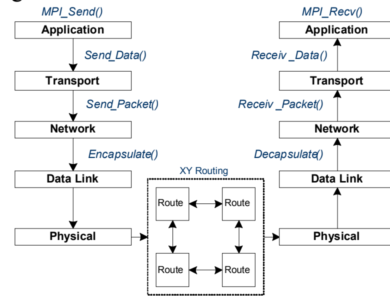

Figure 5 HS-Scale protocol stack

No explicit group synchronization primitives are provided; however this can be simply achieved in an ad-hoc fashion through using MPI_send() and MPI_recv() for passing tokens. Broadcast, gather and similar mechanisms can also be implemented in the same manner. Furthermore, although it could be easily implemented, no non-blocking receive method is provided since the targeted applications usually do not require it.

The prototypes of those functions are as follows:

```
MPI_Send(int edge, const void *data, int size)
MPI_Recv(int edge, void *data, int size)
```

The prototypes of these functions are self explanatory, a reference to the graph edge identifier, a constant void pointer and a size of data expressed in bytes.

Figure 6 shows an example of task graph where it can be seen that communication channels feature a (software) FIFO queue at the receiver side. Queues sizes can be parameterized and their size can be tuned on-line as the operating system provides memory allocation and deallocation services.

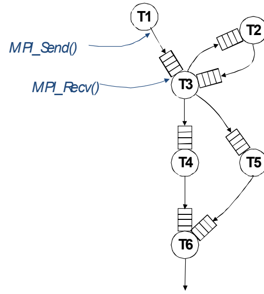

Figure 6 Example of graph task

5. Adaptive task migration

HS-Scale is a complete scalable framework: the proposed MPSOC system is made of a homogeneous set of processing elements, each one running a micro kernel. The applications can be programmed with some MPI primitives providing a higher flexibility in terms of task mapping and potentially higher performances due to the underlying parallelism. Our objective is now to implement a full adaptive task migration support to take benefit from this potential flexibility and performance. The first challenge was to enable dynamic code loading on a system devoid of any MMU. Then, it was necessary to implement a robust migration protocol allowing to improve intelligently the global performance of a given application running on HS-Scale.

One of the objectives of this work is to enable dynamic load balancing which implies the capability to migrate running tasks from processor to processor. Migrating tasks usually implies:

- To dynamically load in memory and schedule a new process
- To restore the context of the task that has been migrated

5.1. Dynamic process loading

Both points are challenging for such microprocessor targets since, for density reasons, no Memory Management Unit (MMU) is available. A MMU, among other tasks, usually performs the translation between virtual and physical addresses and therefore permits to load and run a code in an arbitrary region of the physical memory.

A possible alternative for enabling the loading of processes without such mechanisms relies on a feature that is partly supported by the GCC compiler that enables to emit relocatable code (PIC: Position Independent Code). This feature generally used for shared libraries generates only relative jumps and accesses data locations and functions using a Global Offset Table (GOT) that is embedded into the generated ELF [15] file. A specific post-processing tool which operates on this format was used for reconstructing a completely relocatable executable. Experiments show that both memory and performance overheads remain under 5% for this solution which is clearly acceptable.

5.2. Task context migration

The migration of a process implies both instantiating a new executable into the memory but also restoring its context. Again, the lack of MMU makes this task difficult since the context of the process includes the stack which embeds not only data (such as return values of functions) but also return addresses that are memory-location dependent. The solution we developed is based on defining migration points that are at specific locations in the code, namely whenever a communication primitive is called. This method is restrictive since it assumes that the computation relies on a strict consumer / producer model where no internal state is kept from iteration to iteration. This translates that there cannot be any dependencies between two adjacent computed data chunks.

When a task migration order is issued by the operating system, the following sequence of action is initiated between NPU1 which is current host for the task and NPU2 which is the future.

- Task runs on NPU1 until it reaches the next communication primitive which freezes execution; task context is then saved into the corresponding Task control Block (TCB) by the operating system. This constitutes a migration point.
- NPU1 transfers task binary code and state information (TCB) to NPU2 and deletes task from its own memory. No further incoming data transfers will occur since NPU1 does not answer to communication requests for this task.
- NPU2 then loads the task into memory, creates the necessary software FIFOs, adds the task on TCB table, broadcasts this information on routing table and resumes execution from the migration point.

5.3. Migration policy

Our objective is that the platform has to be capable of taking decisions that relate to application implementation through task placement in order to be self adaptive. Our constraint is that these decisions have to be taken in a fully decentralized fashion in order to ensure platform scalability. Our purpose is to use distributed run-time monitoring mechanisms on each NPU: based on the monitored information, each NPU has to be capable to migrating a task when necessary, following a predefined migration policy.

Each task of an application is connected to its predecessors and successors with software FIFOs (communication queues). In the previous works [2], we have observed on dataflow applications that the performance (in terms of throughput) of a given task and the utilization of its input FIFO are correlated: the FIFO usage tends to grow when its task requires more processing power to compute incoming data. We assume that task requires more processing power because this task shares the same processing resource with other tasks. Based on this assumption, we propose two adaptive mechanisms to improve performance: the first one is migration when a task requiring more processing power shares the same processing resource with other tasks. The second purpose is replication: in this scenario where the processor is too slow for a given task, the

task will be replicated to increase throughput (this will be explored in future works).

Consequently, we have first developed a FIFO monitoring service in our Operating System. Thanks to this mechanism, each NPU knows the FIFO usage for each task running on the system. When the FIFO usage of a given task reaches a pre-defined threshold on a given NPU, this NPU sends a help request to the most appropriated nearest neighbor. This neighbor must offer at least more CPU time to the task so that it ensures a performance gain: the policy will choose in priority the least loaded neighbors.

In order to illustrate the task migration policy, Figure 7 shows a simple example. Initially, we assume two tasks of a given application hosted onto the same NPU and therefore sharing equally 50% of CPU time (time sliced execution). In this example, the FIFO monitoring service indicates that the FIFO usage of task 2 is above a threshold fixed at 80%. Task 2 requires more CPU time than the NPU can provide. This NPU sends a request for assistance as described in the previous paragraph. As soon as a new host is found, a process is executed for migrating task code, using the protocol explained in the previous section. As depicted in Figure 7.b, once the migration process is completed, the task makes use of a higher percentage of the CPU resources (85%) in order to process the task 2 as fast as required by its incoming data.

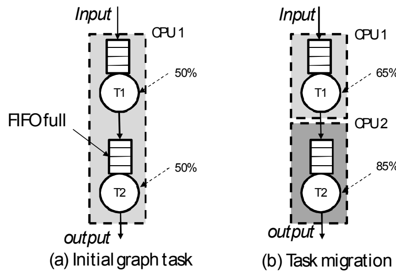

Figure 7 Principle of task migration

6. Experiments and Results

We have proposed a full adaptive migration support based on MPI and implemented it in the HS-Scale framework. In order to validate and to conduct some experiments, we chose to develop a multi-FPGA prototyping platform described in the following. In the second part of this section, we focus on a MJPEG case study to show the feasibility of our adaptive migration support and to measure the dynamic behavior of the system.

6.1. Prototyping platform

The Xilinx Starter Kit Spartan-3 S1000 appeared to fit our specifications in terms of available logic, memory and I/Os for NPU connections. It is essentially based on a Spartan3 S1000 FPGA which 1920 configurable logic blocks (CLB). The board features several general purpose I/Os, 1MByte of fast asynchronous SRAM, several ports for debugging/monitoring purposes and three 40-pin expansion connectors for the interconnections of boards.

Device	# CLB
NPU	624
Router	171
MIPS R3000	366
Other	88

Table 1 FPGA synthesis result

Table 1 gives the device utilization figures for a single NPU hosted on a single board. The complete prototype is an array composed of several instances of the prototyping boards connected through the 40-pin expansion connectors. For debugging reasons, we limited the frequency to 7MHz. One board, i.e. one UART of a single NPU, is directly connected to a PC as depicted on the picture in the Figure 8. This PC is used as a human-machine interface for sending program data (i.e. task codes and microkernel code), the data to compute and to display debugging messages in the monitoring terminal.

Figure 8 Array of 4x4 NPU multi board.

6.2. MJPEG Case Study

The prototyping platform presented above has been used to perform several validations of dataflow applications such as a Finite Impulse Response (FIR) filters, Data Encryption Standard (DES) encoder, and Motion JPEG (MJPEG) decoder [16]. In our previous works [1, 2], we have measured application throughputs in various contexts: task graph running on a single NPU, task graph running on several NPU, several applications running at the same time, etc.

In this paper, our objective was first to validate our adaptive migration process and then to characterize the dynamic behavior of the system when performing task migrations automatically.

We have chosen to study the case of the MJPEG decoder application partitioned into 3 tasks: IVLC, IQ and IDCT. Figure 9 depicts the temporal evolution of the application throughput and the FIFO usage of each task.

During the very first seconds, all tasks are instantiated manually on the NPU(1,1). From t1=4.401s to t2=6.303s, these tasks are executed sequentially on the same NPU which provides an average throughput of 38KB/s. At t2, the IVLC FIFO reaches a value greater than the fixed threshold (80%): this leads to a migration process composed of the following steps:

- checking a migration point (3.79ms)
- looking for a free NPU (5.59ms)

- migrating the task (117.95ms)
- restoring the context (4.02ms)

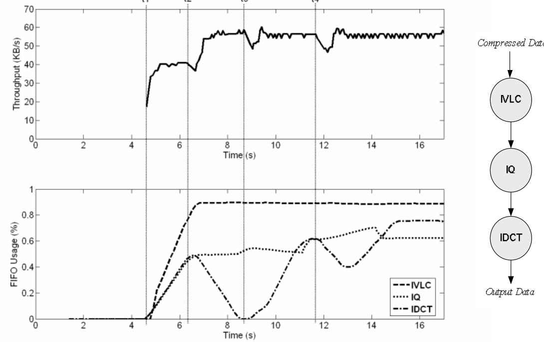

Figure 9 MJPEG execution in time

In this scenario, the task is moved from NPU(1,1) to NPU(2,0): the whole task migration process takes 131.35ms. During this time interval, the OS consumes CPU time to perform the migration process which decreases the application throughput as depicted in the Figure 9. After the migration completion, the average throughput reaches 57KB/s: it takes 87.7 ms to observe a performance benefit.

Observing the FIFO usage reveal that (i) the IQ FIFO remains stable meaning that it has just enough CPU time to process its data, (ii) the IDCT FIFO slowly decreases meaning that it has enough CPU time to process all the data in its FIFO. In this situation, the mapping is stable from the migration policy point of view: this implies that the system won't move the tasks anymore.

In order to observe what would be the consequence of another migration, the user has the possibility to initiate intentionally a new migration. In order to completely unfold the whole task graph, at t3=8.706s, the IQ task is migrated by the user from NPU(1,1) to the NPU(2,1): this migration lasts 78ms. During this period, we observe again a short decrease in the application throughput. The average throughput stabilizes after a short period around its previous value. This proves that actually the performance in terms of throughput cannot be improved anymore because IVLC is the critical task of the pipeline: the global application throughput is completely dependent on the IVLC throughput. The IVLC FIFO being still greater than 80% and running on a single NPU, the only way to increase the application throughput would be either to increase the NPU frequency or to replicate the same task on another NPU. From a FIFO point of view, one can also notice that the IDCT FIFO increases: after migration from NPU(1,1), the IQ task is running alone onto NPU (2,1) which provides more CPU time. After 3s, the FIFO utilization stabilizes again.

At t4=11.769s, the user initiates another migration of the IQ task to the NPU(1,2): this migration lasts 76ms. The effect on the application throughput is similar to the previous one. However, we observe a decrease of the IDCT FIFO usage as IQ does not produce new data during the migration time. After 3s, the system finds again a stable state in terms of throughput and FIFO usage.

7. Conclusion

There are several approaches for MPSOC, but homogeneous solutions are attractive because of their natural scalability. HS-Scale is our contribution in this domain. In this paper, we have proposed an original MPI-based adaptive migration support for the HS-Scale system. Each NPU is able to trigger a migration: it depends on the usage of the tasks' FIFOs monitored by a dedicated service of the OS and a predefined policy. This mechanism is completely distributed, ensuring the scalability of our approach. It has been validated on a multi-FPGA prototyping platform and a case study on the MJPEG decoder has illustrated the dynamic behavior of the system. One very interesting observation is that the system is able to find itself a stable solution providing the best application throughput.

In our future works, we will focus first on different case studies. Especially, we would like to measure the behavior of our system when running several applications. Our major concern is also to increase the self-adaptability in term of performance: the automatic replication of tasks will be particularly studied and will be compared to alternative solutions such as Dynamic Frequency Scaling.

8. References

[1] "HS-Scale: a Hardware-Software Scalable MP-SOC Architecture for embedded Systems". Saint-Jean N, Sassatelli G, Benoit P, Torres L, Robert M. VLSI, 2007. IEEE Computer Society Annual Symposium on Volume , Issue , 9-11 March 2007 Page(s):21 - 28

[2] "Application Case Studies on HS-Scale, a MP-SOC for Embbeded Systems". Saint-Jean N, Benoit P, Sassatelli G, Torres L, Robert M . IC-SAMOS 2007.(July 2007) pp 88-95

[3] "MPCore Linux 2.6 SMP kernel and tools", ARM Limited, www.arm.com/products/CPUs/linux2 6 smp.htmlSdgsdf

[4] "Locality-Aware Process Scheduling for Embedded MPSoCs", M.T. Kandemir, G. Chen, Proceedings of DATE, pp. 870–875, 2005.

[5] "Multi-Processor Operating System Emulation Framework with Thermal Feedback for Systems-on-Chip". Salvatore Carta, Michele Pittau, Andrea Acquaviva, Pablo G. Del Valle†, David Atienza, Giovanni De Micheli, Fernando Rincon, Luca Benini, Jose M. Mendias. Great Lakes Symposium on VLSI 2007 pp 311 - 316.

[6] "Supporting Task Migration in Multi-Processor Systems-on-Chip: A Feasibility Study". Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. DATE 2006 pp 1- 6

[7] "Mparm: Exploring the MPSoC design space with SystemC". L. Benini, et al. Journal of VLSI, September 2005. Volume 41, Number 2 / septembre 2005 pp 169-182

[8] "Scalable Cluster Computing with MOSIX for Linux". Barak A., Laadan O. and Shiloh A., Proc. Linux Expo '99, pp. 95-100, Raleigh, N.C., May 1999.

[9] "A Task Migration Implementation of the Message-Passing Interface". Jonathan Robinson, Samuel Russ, Bjorn Heckel, Brian Flachs. HPDC 1996 pp 61

[10] "Improving load balancing in an MPI environment with resource management". . A. R. Dantas and E. J. Zaluska. Springer Berlin / Heidelberg 1996 High-Performance Computing and Networking pp 959-960

[11] http://t-system2.polnet.botik.ru/checkpointing/18-Youhui_Z.pdf

[12] "A Grid Middleware for Distributed Java Computing with MPI Binding and Process Migration Supports". Lin Chen, Cho-Li Wang, Francis C.M. Lau, and Ricky K. K. Journal of Computer Science and Technology volume 18 , issue 4 (July 2003) pp 505 – 514

[13] MIPS corp., http://www.mips.com

[14] "Hermes: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip". Moraes F. et al. Integration the VLSI Journal 38, October 2004. pp. 69-93.

[15] "Linkers and Loaders" by John R. Levine, published by Morgan-Kauffman in October 1999

[16] www.wikipedia.org/wiki/JPEG