Architectural Support for Task Migration Concerning MPSoC

Alexandra Aguiar[1], Sérgio J. Filho[1],
Tatiana G. dos Santos[2], César Marcon[1], Fabiano Hessel[1]

[1]Faculty of Informatics – PUCRS – Porto Alegre, Brazil

[2]University of Santa Cruz do Sul – UNISC – Santa Cruz do Sul, Brazil

{alexandra.aguiar, sergiojf},@inf.pucrs.br,

tatianas@unisc.br, {cesar.marcon, fabiano.hessel}@inf.pucrs.br

Abstract. *Embedded computing systems are currently present in a wide range of consumer goods and their main characteristic is the implicit specialized behavior but keeping a certain level of flexibility once it avoids redesigning due to small requirement changes. Thus, microprocessors usually are a good alternative to achieve such flexibility. Consequently, embedded systems designs make use of homogeneous or heterogeneous processors families for its complete implementation, which is known as Multiprocessor System-on-Chip (MPSoC), which can have a performance speed up through using dynamic load balancing strategies, such as task migration, to fairly distribute the existing tasks among all embedded processors. The objectives of this work are to discuss architectural aspects for embedded systems, which allow a dynamic task migration and its implications, as well as to present different techniques for the dynamic task migration showing their possible use in MPSoCs.*

1. Introduction

In embedded computing systems, ordinarily, the execution of the tasks should be performed in a given period, thus imposing time restrictions. Systems, which present such type of restriction, are known as *real-time systems* (RTSs) and make use of a specific operating system known as *Real-Time Operating System* (RTOS) to make their management in a higher level of abstraction viable. Besides other OS common features, one of the main purposes of the RTOSs consists in trying to guarantee that a task execution is completed respecting its timing restrictions, which usually is responsibility of the RTOS task scheduler [Farines et al. 2000].

On the other hand, due to cost and performance issues, it is desirable to implement embedded systems in a single chip *System-on-Chip* (SoC), which allows the use of heterogeneous components such as CPUs, memories and buses. Furthermore, it is possible that a SoC is composed by more than one processing element (PE), being known as *Multiprocessor System-on-Chip* (MPSoC). Thus, the requirements and implementation characteristics needed by a given application must be taken into account during the development of an MPSoC causing that customized architectures be designed [Jerraya et al. 2005]. In this context, the need for a load balance of the system can be observed in MPSoCs as well as in general purpose distributed and parallel computing systems. This issue has been previously and broadly studied in these systems' areaand with the rise of the MPSoCs, it is object of research in both academia and industry [Bertozzi et al. 2006].

The main goal of this work consists of presenting different alternatives of architectural support for dynamic load balancing techniques, such as task migration, concerning real-time MPSoCs. Through using such techniques, it might be possible to distribute dynamically and homogeneously the system's load, avoiding execution overload. It is mainly due to execution overloads that many permanent failures of the system occur, like electro migration, stress migration and dielectric breakdown [Council 2006]. Avoiding overloaded spots may increase the application performance as well as delaying different types of failures.

The remainder of the paper is organized as it follows. The next section shows the related work and points their importance to this paper. In Section 3 several issues regarding to load balancing are presented. Section 4 shows some architecture proposals regarding to load balancing and particularly task migration evaluations. Finally, Section 5 concludes this work and presents some future work.

2. Related work

Although load balance techniques for general purpose parallel computing have already been studied in some previous works [Suen and Wong 1992], [Chang and Oldham 1995], [de Mello and Senger 2006] let's focus here on the works concerning the embedded computing issues. Is spite of having some similarity with general purpose parallel computing, once both architectures have multiple processing elements, MPSoCs present many different challenges within their own design time, which makes even more difficult to implement efficient task migration mechanisms.

Nollet et al. [Nollet et al. 2005] propose a reuse technique, concerning the processor's debug registers, in order to decrease the initial overhead of a heterogeneous MPSoC task migration. Therefore, the OS must verify the system workload and notify the task that should be migrated. It is important to notice that, when dealing with task migration, specially on embedded systems, one of the main concerns must be the way to access memory, once one should consider that task migration implies in transferring, somehow, both task code and data from a node to another.

Streichert et al. [Streichert et al. 2006] on task mapping optimization with fault tolerant mechanisms. The main idea here is to guarantee, once a fail occurs, that the executing tasks are migrated to other non-fault PE. In this work, the architecture details are not presented although the presented model for the task biding considering fault tolerance could be used as threshold criteria to initiate the task migration process supported by the architecture to be discussed throughout this paper.

Bertozzi et al. [Bertozzi et al. 2006] present an approach that deals with MPSoCs task migration. They propose a strategy where the user is responsible for setting the possible migration points in the application code. The architecture of the authors' work is composed by one master and an arbitrary number of slaves cores. Even though this paper presents an architecture that could be used as a basis to our work its need for migration points is not desired at all once our work is more focused on achieving a architectural support for dynamic task migration.

Götz et al. [Götz et al. 2007] present a design flow for dynamic relocation of hybrid tasks. These tasks may be executed either in hardware or in software and are represented through a *state transition graph*, where each state is known as computation block and stands for a given task operation. In our opinion as important as the architecture support for the task migration, the consideration of the correct threshold migration parameters are vital to the feedback of successful results in migration performances.

Barcelos et al. [Barcelos et al. 2007] propose a hybrid memory organization approach, which puts together both centralized and distributed memory organizations focused in a task migration with less energy consumption. Therefore, depending on the position of the source and destination node in the Networks-on-Chip (NoC) architecture, the data to be migrated is taken either from this source node or from a global shared memory. In the same research group Brião et al. [Brião et al. 2007] takes into account the task migration overhead in a dynamic environment and discusses its impacts in terms of energy, performance and real-time constraints for NoC based MPSoCs. Studying the impact of task migration is especially important when talking about load balancing systems, once it enables the load balancing technique.

Coskun et al. [Coskun et al. 2007] show a task scheduling system aware of the MPSoC temperature. Indeed the die temperature might be one of the greatest motivations for implementing load-balancing techniques once hot spots accelerate permanent fault mechanisms such as the previously mentioned electro migration, stress migration and dielectric breakdown [Council 2006]. Having such information, it might be possible to decide migration threshold situations to performance, energy consumption and related with die temperature measurements.

3. Load balancing

Load balancing issues have been addressed for many years in general purpose distributed and parallel systems [Casavant and Kuhl 1988], [Suen and Wong 1992], [Chang and Oldham 1995], [de Mello and Senger 2006]. Considering it, it is important to introduce the main concept of load balancing, which is the division of the amount of work among several nodes of a given group of processing nodes. The main objective is to guarantee that none or at least the minimum possible number of the nodes neither are overloaded nor underloaded. The main advantages in assuring a balance of the total system workload are (i) having a good utilization of all the system nodes, (ii) improving overall performance and (iii) minimizing communication delays.

According to [Casavant and Kuhl 1988], load-balancing mechanisms can be classified into either global or local policies. Local policies concern individually each node of the system and its own scheduling issues, while global one takes into account decisions such as where to execute a process in the available nodes. Besides, there is other main part of the taxonomy, which considers both static and dynamic policies. They regard the time at which the scheduling or assignment decisions are made.

Static load balancing is a method in which depending on the nodes' performance the workload is distributed in the beginning of the system execution. The main advantage of this method is the communication reduction between load controller and the other components, which boosts the performance. Conversely, the main drawback is that it cannot adjust itself to the runtime unknown issues, thus decreasing the application performance [Abubakar and Aftab 2004].

On the other hand, dynamic load balancing determines the distribution of workload at run-time. The master of the system performs the assignment of new tasks to the remaining members, depending on the recent information collected. Since the workload distribution is done during runtime, it may give better performance at the cost of overhead associated with communication needs [Abubakar and Aftab 2004]. Therefore, the overhead associated should be limited in a reasonable way in order to achieve better performance [Dandamudi 1998]. There are many dynamic load-balancing algorithms proposed. They have four basic steps in common [Zaki et al. 1996]:

- *Load monitoring*, that is monitoring system's components performance;
- *Synchronization*, that is exchanging this information among nodes;
- *Rebalancing criteria*, that is calculating new distributions and making the work movement decision;
- *Task migration* or *actual data movement*.

Regarding to current OSs, like the Linux 2.6 kernel, it is noticeable their capability of utilizing the power offered by a shared memory architecture, either loosely or tightly coupled. The key feature consists in the ability to balance workload across the available CPUs while maintaining cache efficiency, which can be compromised because when a task is associated with a single CPU, moving it to another CPU requires the cache to be flushed. This will increase the latency of the task's memory access until its data is, indeed, in the cache of the new CPU. That is why the kernel keeps a pair of runqueues for each processor and each runqueue supports a given number of priorities, with a top number, which is used for real-time tasks, and the rest for user tasks. Then, time slices are given to tasks for their execution and after using their allocation of time slice, they are moved from the active runqueue to the expired runqueue. This mechanism provides fair access for all tasks to the CPU. In that case, with a task queue per CPU, workload can be balanced given the measured load of all CPUs in the system. This measurement is given by the scheduler, which performs load balancing to redistribute the tasks along the nodes.

In RTOS systems, load-balancing techniques are desirable due to the same advantages presented so far. In this context, one important thing is to maintain transparency of the load balancing technique and one method for achieving it is to configure the RTOS, so that individual threads can be assigned to run on specific processors based on its availability. Doing so, the processing load can be shared among processors with work automatically assigned to a free processor. The RTOS must determine whether a processor is free and, if it is true, a thread can be run on that processor even though the remaining of the system may already be running other threads. In addition, priorities are also important to consider once the RTOS scheduler is designed to maintain priority execution of all threads, in such way that higher priority threads are executed before lower priority threads. Priority-based, preemptive scheduling uses multiple cores to run threads when they are ready. The scheduler automatically runs threads on available cores. Details regarding to our proposal on the scheduler's implementation will be detailed along the Section 4.

Consequently, it is possible to see that automatic load-balancing feature is beneficial to overall system performance. For instance, one processor can be the responsible for all external interrupt handling, which leaves the other processor free to focus in its application processing, even during periods of intense interrupt activity that could degrade performance. In the following section, load balancing monitors strategies are presented. These approaches serve as base for the implementation of dynamic load balancing techniques, which are supported in the forthcoming architectures' discussion.

3.1. Load balancing monitors

In order to get a fairly balanced system, monitors responsible for analyzing the load of the system are desirable and there are two main approaches concerning their implementation which are centralized and distributed ones. Both concepts are based in [Lan and Yu 2001] definitions.

In a centralized load balancing system, a single node collects the global load information. The other nodes send their load status messages to the central scheduler and all load-balancing decisions are made at the central scheduler based on the collected messages. A main issue of centralized load balancing mechanisms is relatively low reliability, once its failure leads to load balancing policy disoperation.

Whereas, in a decentralized load balancing system, each node broadcasts its load information periodically to other nodes, so they can update their own load tables. Every node performs its node selection based on the global system load status, which is continually obtained. The major drawback is that every node, including busy ones, must keep track of incoming update messages, as well as generate update messages of its own. Finally, taking the load balancing as a background, the next section discusses the proposed architectures for dynamic load balancing mechanisms through task migration in real-time MPSoCs.

4. Proposed architectures

In this section, several architectures are proposed in order to allow load balancing in MPSoCs through task migration. Initially, the base platform is discussed followed by the architecture strategies.

4.1. Base platform

The platform is composed of one Plasma [Cores 2007] processor core, which implements the MIPS I instruction set. Several advantages arise in the use of this processor, such as: open source implementation, ease integration among multiple cores through some interconnection layer, pre-developed compilers and reasonable performance for most of embedded applications. Besides the processor core, other components complete this basic architecture, such as a netlist of the processing core, an internal memory, an UART and an interface to external memory. All these components communicate through an internal bus. A testbench, which generates clock and resets signals to the processor core and other modules, was created, allowing the simulation of the platform functionality in an RTL simulator.

To create an MPSoC platform we used four Plasma processors like the one described earlier and their communication is done through a media control access and a bus. The implemented architecture is presented as a block diagram depicted in Figure 1.

It is important to notice that although the interfaces are defined in a very generic way, other communication strategies, like NoC, can be adopted instead. Regarding that it

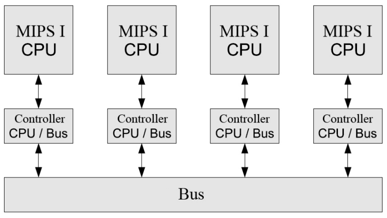

Figure 1. MPSoC base platform to be used in the study

is a simple system, where there are no DMA controllers and the processor is the responsible for managing all the communication, it is reasonable to use a smaller logic communication solution, once without the proper support, a more efficient communication method would be underutilized anyway.

Moreover, this platform was implemented in VHDL and prototyped in FPGA. Several software communication drivers were developed, using simple *send* and *receive* blocking and non-blocking primitives. Besides, other operating systems can be used in this architecture. The main options regarding this study are:

- EPOS [Fröhlich and Schröder-Preikschat 1999] - an OS oriented to application, which adapts itself to the user application requirements;
- PlasmaRTOS [Cores 2007] - OS available for execution on Plasma processor;
- Own developed *micro-kernel*, which is a small OS, when compared to the previous ones.

4.2. Base platform and monitors

In order to supervise dynamically the system load, the monitor strategy is a main concern. Therefore, analyzing the existing approaches (previously reviewed in Section 3.1) and considering the base platform, it is possible to propose three different monitors strategies here.

The first approach is a *hardware-centralized monitor*, which is implemented as a dedicated hardware module. It communicates with the processors, storing their workload information. New tasks arrive, for instance, when the monitor distribute them and evaluate their impact on the entire system. This strategy is depicted in part A of Figure 2, where can be seen the four processors connected through a bus and a dedicated hardware load monitor. Another possible approach concerns to a *software centralized monitor*, that is, a dedicated processor acting like a load monitor. Similarly to the hardware centralized approach, this monitor should store the state of the system workload to take the different decisions regarding to a possible migration of tasks. This approach is shown in part B of Figure 2 among the other processing elements being all of them connected through a bus.

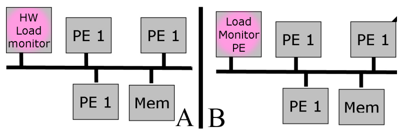

Figure 2. Centralized hardware (A) and software (B) load monitor

The last proposed type of architecture is a *distributed monitor strategy*, in which the operating system of each Plasma core should always try to agree with each other in relation to the total system workload. When a processor is overloaded, it might ask for others help. If one is underloaded, it can automatically send its load to a normal loaded one and put itself to some sort of sleeping mode. Figure 3 shows this strategy, where the processors come to an agreement themselves in order to keep the system balanced.

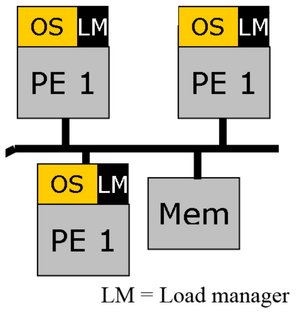

Figure 3. Distributed load monitor

The main benefits of this implementation concerns its level of abstraction; it can be done at OS level being easily updated or extended. Besides, neither the silicon area of the chip needs to increase nor a dedicated module needs to be implemented. Overall, this implementation is independent of both the communication architecture used and of possible updates in the PE hardware. Finally, as it is a naturally distributed strategy, this type of monitor should not affect significantly in the overall MPSoC performance.

No matter what monitor strategy is adopted, it should evaluate other issues than just system workload. One of these concerns could regard to some temperature sensors, for instance, in order to keep the chip temperature also balanced. Energy consumption of each core could also be taken into account and specially be evaluated against the pure workload information, thus generating a special type of monitor, which tries the best workload taking into account energy consumption issues.

4.3. Task migration

When load-balancing strategies are adopted, they must define the way the load is actually going to be switched among the different processing elements. One of the main approaches regarding this is the task migration one that, somehow, must migrate tasks that are being executed in a given overloaded or underloaded node to an average loaded one.

Regarding to some embedded systems own issues like commonly limited memory, performance requirements and energy saving concerns, we propose the use of two different levels of hierarchy in order to perform task migration without the degradation of the systems performance at all.

The first instance of the hierarchy is a set of processors with shared memory, so that for task migration to happen the processors need simply send the task identification to the recipient, which can easily access all needed task information through the shared memory. This type of migration is here defined as *internal migration*. In this case, each set of processors would have a monitor strategy implemented allowing the set to be fairly balanced. Figure 4 shows this strategy in two different parts regarding to the monitors' strategy we believe can better fit to it. The idea of keeping sets of processors is that they are small, so that centralized monitors (either hardware or software) could be implemented without compromising scalability.

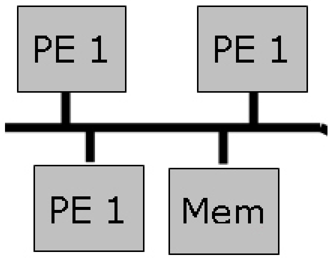

Figure 4. A single set of processor: internal migration is allowed

The second part of the system is formed by several sets of processors with no shared memory strategy. In this case, in a distributed way, sets of processors could agree among themselves about migrating tasks from one given set of processor to another. In this case, all the information concerning the task should also be sent and this overhead should be included when considering if migration is needed or not. This type of migration is defined to be *external migration*. As the number of sets of processors might increase once they will not have shared memory limitations, we believe that the distributed monitor is more indicated and this is presented in Figure 5.

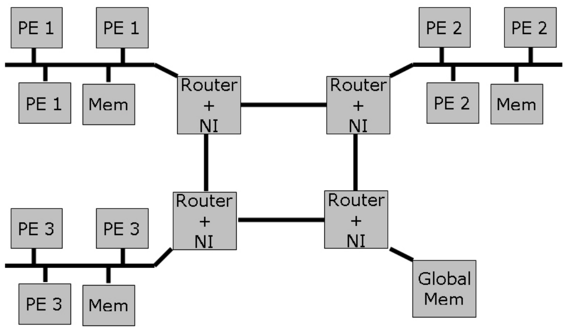

Figure 5. Several sets of processors: external migration is allowed

In a certain way, this approach is locally symmetric multiprocessing (SMP) and globally asymmetric multiprocessing (AMP) and tries to join advantages of both implementation strategies. Preliminary studies point that by doing this kind of implementation, when considering hard real-time embedded systems, it might be possible to avoid deadline missing, especially of hard real-time aperiodic tasks once when this type of task occurs:

- Other best effort tasks might be migrated internally in the same set of processors, making enough space for the execution of the just arrived task;
- If the external migration is not going to violate any of the just arrived task deadlines, it could be performed in order to balance the system workload.

Finally, even regarding to all the potential benefits in using dynamic task migration mechanisms in real-time MPSoCs, its effective use is not a reality yet. Indeed, the very MPSoCs design issues can be faced as a series of potential to such implementations [Bertozzi et al. 2006]. Thus, these challenges have not allowed yet the full implementation of a total dynamic task migration mechanism in RTL level concerning this type of systems.

5. Conclusion and Future work

Embedded systems are massively present in people's lives and each more advances in technology allow their growing impulse. Thus, MPSoC are being commonly used to implement such systems, hence performance requirements and energy constraints might be addressed. Besides that, embedded applications might also present real-time constraints increasing the system design and management complexity.

Although MPSoCs tend to be dedicated to their application, one can notice its resemblance with general-purpose parallel computer in the sense both have several processing elements connected through some channel. For this reason, former and classical issues of general-purpose parallel and distributed system are again being topic of research and load balancing can be highlighted in this context.

In this paper, load-balancing theory is taken as background to achieve the main objective of this work: propose architectures, which are able of implementing load balancing mechanisms and thus, allowing dynamic task migration among several nodes of an MPSoC. The preliminary prototype was detailed and other studies regarding to the load balancing monitor implementations were addressed. Future work includes the use of existing and functional distributed applications in the platform with the proposed architecture updates and the comparison between dynamic load balancing system and regular system strategies.

References

Abubakar, H. R. and Aftab, U. (2004). Evaluation of load balancing strategies. In *Proceedings of National Conference on Emerging Technologies*.

Barcelos, D., Brião, E. W., and Wagner, F. R. (2007). A hybrid memory organization to enhance task migration and dynamic task allocation in NoC-based MPSoCs. In *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*, pages 282–287, New York, NY, USA. ACM.

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

Brião, E., Barcelos, D., Wronski, F., and Wagner, F. R. (2007). Impact of task migration in NoC-based MPSoCs for soft real-time applications. In *IFIP INTERNATIONAL CONFERENCE ON VERY LARGE SCALE INTEGRATION*, New York, NY, USA.

Casavant, T. L. and Kuhl, J. G. (1988). A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154.

Chang, H. W. D. and Oldham, W. J. B. (1995). Dynamic task allocation models for large distributed computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(12):1301–1315.

Cores, O. (2007). Plasma most MIPS I(TM) opcodes. http://www.opencores.org.uk/projects.cgi/web/mips/, Available at <http://www.opencores.org.uk/projects.cgi/web/mips/>. Accessed at 01 dez.

Coskun, A. K., Rosing, T. S., and Whisnant, K. (2007). Temperature aware task scheduling in MPSoCs. In *Design, Automation and Test in Europe Conference and Exhibition, 2007. DATE '07*, pages 1–6.

Council, J. E. D. E. (2006). Failure mechanisms and models for semiconductor devices. www.jedec.org/ download/search/jep122C.pdf.

Dandamudi, S. P. (1998). Sensitivity evaluation of dynamic load sharing in distributed systems. *IEEE Concurrency*, 6(3):62–72.

de Mello, R. F. and Senger, L. J. (2006). Modelo de migração baseado na avaliação da carga e tempo de vida de processos em ambientes heterogêneos. *IEEE Latin America Transactions*, 4(5).

Farines, J.-M., da Silva Fraga, J., and de Oliveira, R. S. (2000). *Sistemas de Tempo Real*. Second Escola de Computação, IME-USP, São Paulo-SP.

Fröhlich, A. A. and Schröder-Preikschat, W. (1999). High performance application-oriented operating systems, the EPOS aproach. In *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, pages 3–9.

Götz, M., Xie, T., and Dittmann, F. (2007). Dynamic relocation of hybrid tasks: A complete design flow. In Sassatelli, G., Glesner, M., Bobda, C., and Benoit, P., editors, *ReCoSoC*, pages 31–38. Univ. Montpellier II.

Jerraya, A., Tenhunen, H., and Wolf, W. (2005). Multiprocessor systems-on-chips. *Computer*, 38(Issue 7):36– 40.

Lan, Y. and Yu, T. (2001). A dynamic central scheduler load balancing mechanism. *IEEE Trans. on Parallel and Distributed Systems*, 12(9):899–911.

Nollet, V., Avasare, P., Mignolet, J.-Y., and Verkest, D. (2005). Low cost task migration initiation in a heterogeneous MP-SoC. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 252–253, Washington, DC, USA. IEEE Computer Society.

Streichert, T., Strengert, C., Haubelt, C., and Teich, J. (2006). Dynamic task binding for hardware/software reconfigurable networks. In *SBCCI '06: Proceedings of the 19th annual symposium on Integrated circuits and systems design*, pages 38–43, New York, NY, USA. ACM.

Suen, T. and Wong, J. (1992). Efficient task migration algorithm for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 3(4).

Zaki, M. J., Li, W., and Parthasarathy, S. (1996). Customized dynamic load balancing for a network of workstations. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 282, Washington, DC, USA. IEEE Computer Society.