COMMUNICATION-AWARE COMPONENT ALLOCATION ALGORITHM FOR A HYBRID ARCHITECTURE *

Marcelo Götz, [1] Achim Rettberg [2] and Carlos Eduardo Pereira [3]

[1]*Heinz Nixdorf Institute*
University of Paderborn, Germany
mgoetz@uni-paderborn.de

[2]*C-LAB*
University of Paderborn, Germany
achim@c-lab.de

[3]*Electrical Engineering Department*
UFRGS, Brazil
cpereira@ece.ufrgs.br

Abstract High computational performance and flexibility are the requirements of nowadays embedded systems and they are increasing constantly. A single architecture must be able to support different application with dynamically requirements (changing environments). As an operating system (OS) is desired to provide support for such systems, it has to use the available resources in an optimal way (competing with the application), since an embedded system architecture usually lack in resources. Therefore, we present here our approach towards a reconfigurable RTOS that is able to distribute itself over a hybrid architecture (comprising FPGA and CPU). In this paper we will concentrate in the strategies used to allocate the OS services over a hybrid architecture, taken into consideration the used resources in the running domain (CPU or FPGA) and the communication costs.

Keywords: Reconfigurable Computing, System-on-Chip, Real Time Operating System

*This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

1. INTRODUCTION

Embedded systems are increasingly requiring more computational performance and flexibility due to the growing application complexity and changing environments where these systems are inserted. Additionally, the used execution platforms are usually lacking in resources, which make the instantiation of a complete system a challenge for a system developer.

In counterpart, the development of an execution platform for such systems may profit from modern Field Programmable Gate Arrays (FPGAs), like the Virtex-II Pro, where a CPU is hardcore embedded into the fabric. Thus, high computation performance can be achieved by implementing components in hardware. Additionally, flexibility is provided due to availability of a CPU and the partial reconfigurable capability of such devices.

In order to easier the activities of the user when developing the application, the used Operating System (OS) needs to tackle the underlying platform properly. Actually, the reasons to use an OS for a Reconfigurable Systems-on-Chip (RSoC) are not different from those for running an OS on any system ([3]).

Towards this objective, we are developing a reconfigurable RTOS that is able to distribute itself over a hybrid architecture, which comprises a CPU and a FPGA. In changing environments, where application requirements are dynamic, the RTOS needs to provide the services currently needed by the running application. However, due to the lack of resources of the underlying platforms, a complete instance of a RTOS is usually not possible. Therefore, we propose to reconfigure the RTOS at run-time over the hybrid architecture in order to better use the available resources, which is also shared by the application tasks.

Our proposal fits into the scope of an in-house ongoing research, where support for self-optimizing systems is being studied. For such systems, a self-optimizing RTOS is also required. In this paper, however, we will concentrate on the strategies used to allocate the OS services over the hybrid architecture, taken into consideration the used resources in the running domain (CPU or FPGA) and the communication costs.

The remaining of this paper is organized as follows. Section 2 describes the related work, followed by a detailed discussion of the RTOS we are using (Section 3) including a briefly description of the previous OS service allocation algorithm. In this section, we also describe the execution platform. We then present our communication-aware allocation algorithm is Section 4. There, the clustering and the allocation of the components are described. In Section 5 we present the evaluation results and we finalize in Section 6 with our conclusions.

2. RELATED WORK

The overhead added by the operating system used for embedded systems need to be carefully considered due to the usual lack of resources provided by the underlying platform. However, up to now all approaches have been based on implementations that are static in nature, see [9], [8], [7], [10] and [11]. It means that they do not change at run-time, even when application requirements may significantly change.

Reconfigurable hardware/software based architectures are very attractive for implementation of run-time reconfigurable embedded systems. The hardware/software allocation of applications tasks to dynamically reconfigurable embedded systems (by means of task migration) allows for customization of their resources during run-time to meet the demands of executing applications, as can be seen in [6].

An example of this trend is the Operating System for Reconfigurable Systems (OS4RS) ([13]). This work proposes an operating system for a heterogeneous reconfigurable System-on-Chip (SoC). It aims to provide an on-the-fly reallocation of specific application tasks, over a hybrid architecture, depending on the computational requirements and on the Quality of Service (QoS) expected from the application. Nevertheless, the RTOS itself is still static. Moreover, the reconfiguration time cost is not a big issue in the design.

Additional research efforts spent in reconfigurable computing field are only focusing on application level, leaving to the RTOS the responsibility to provide the necessary mechanisms and run-time support. The works presented in [1], [14] and [12] are some examples of RTOS services to support the (re)location, scheduling and placement of application tasks on an architecture composed by FPGA with or without an CPU. In our proposal, we expand those concepts and propose new ones to be applied in the RTOS level. Thus, the RTOS can profit from the reconfigurable hybrid architecture in order to make a better usage of the available resources in a flexible manner. Moreover, from our knowledge there are no other works dealing with on-line RTOS services migration between hardware and software execution environments.

Nevertheless, note that in our currently approach, we do not consider that a OS service may be preempted in CPU and resumed at FPGA (or vice-versa). This is different from [13]. In our approach the migration occurs when the OS component is not being used. Beside that, we focus on OS component migration instead of application tasks. In [4] we show how a migration may dynamically be executed without requiring service preemption during migration.

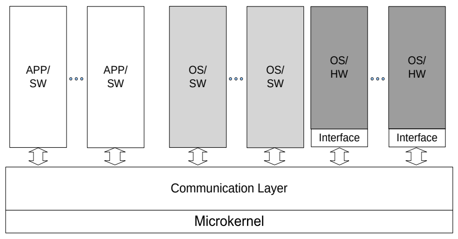


Figure 1. Proposed microkernel based architecture.

3. PRELIMINARIES

Our RTOS is composed of a set of services that may run either on the CPU or on the FPGA. Therefore, the reconfigurable services are provided in two implemented versions: software and hardware. In our approach, most of the application tasks run on the CPU and only application critical tasks use FPGA resources.

The target RTOS architecture follows the microkernel concept, where application and operating system services are seen as components running on top of a small layer which provides basic functionalities. The Figure 1 shows abstractly our architecture. Additionally, the communication infrastructure layer provides the necessary support to allow the communication among components running over the hybrid architecture in an efficient manner. More details about this topic are provided in Section 3.3.

Without loss of generality, we assume that over the hybrid architecture the OS services are seen as components, which uses system resources (FPGA area, CPU workload and communication bandwidth). This view is enforced by the usage of the microkernel architecture model.

3.1 PROBLEM STATEMENT

From a higher point of view, we can see two major problems related with the OS services: their allocation and their reconfiguration. The allocation of the OS services over the hybrid architecture should be done in such a way to optimize the used resources (including the communication costs). This is the focus of the current paper.

Nevertheless, as we considered a changing environment, this allocations need to be continuously evaluated. Whenever the OS components allocations are required to change, a system reconfiguration happens. This reconfiguration activity needs to be carried out respecting the correctness of the running application. In a real-time system, it means that the reconfiguration activities can not violate any time constraint of the running tasks. This problem is handled by modelling theses activities as aperiodic jobs and scheduled together with the running tasks using, therefore, a server ([4]).

3.2 COMMUNICATION UNAWARE ALLOCATION ALGORITHM

A heuristic algorithm presented in [5] determines the allocation OS components. Although presenting good performance, this algorithm does not take into consideration the communication costs. It decides at run-time where to place each OS component taking into consideration its current cost and the remaining available system resources. Here, the resources are: FPGA area (for components being located in hardware) and CPU processor utilization (for components being located in software). Thus, the system has to locate the RTOS components in a limited FPGA area (A_{max}) and limited CPU processor workload (U_{max}).

Every component i has an estimated cost $c_{i,j}$, which represents the percentage of resource from the execution environment used by this component. On the FPGA ($j = 2$) it represents the circuit area needed by the component and on the CPU ($j = 1$) it represents the processor load used by it. The heuristic mentioned above minimizes a objective cost function (Equation 1) subjected to a system resources constraints (Equations 2 and 3).

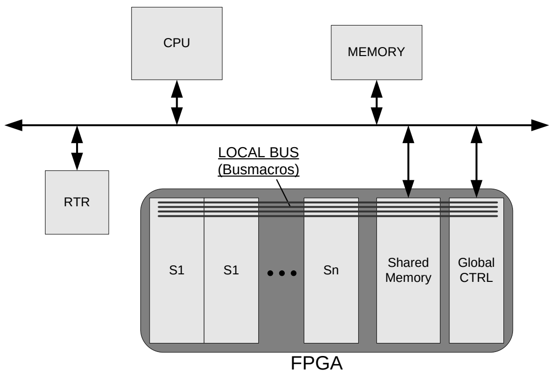
$$min\{\sum_{j=1}^{2}\sum_{i=1}^{n} c_{i,j}x_{i,j}\} \tag{1}$$

$$U = \sum_{i=1}^{n} x_{i,1}c_{i,1} \leq U_{max} \tag{2}$$

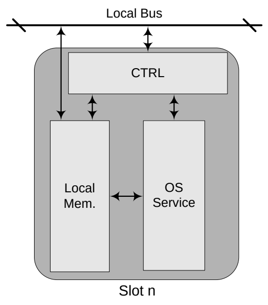
$$A = \sum_{i=1}^{n} x_{i,2}c_{i,2} \leq A_{max} \tag{3}$$

Besides these constraints, an additional one is defined in order to maintain a balanced resource utilization: $B = |w_1U - w_2A| \leq \delta$. Where δ is the maximum allowed unbalanced resource utilization between CPU and FPGA. We also consider that a component i can be assigned just to one of the execution environment. Thus, $\sum_{j=1}^{2} x_{i,j} = 1$ for every $i = 1, ..., n$. The weights w_1 and w_2 are used to proper compare the resource utilization between two different execution environments.

Due to the application dynamism, the assignment decision needs to be checked continuously. Whenever the specified constraint δ is no longer fulfilled, a system reconfiguration takes place. This implies that a set of RTOS component needs to be relocated (reconfigured) by means of migration. In other words, a service may migrate from software to hardware or vice-versa.



(a) Architecture overviwe



(b) OS service slot template

Figure 2. System architecture.

The working algorithm is composed of two phases. First, starting from an empty CPU and FPGA utilization, the components having the smallest costs are selected first and placed on either CPU or FPGA, trying to keep the resource utilization between these two execution domains the same. In the second phase (based on Kernighan-Lin [2]), the allocation is refined by changing the previous location of a component pair (each one locate in different execution domain). This last phase is used in order to improve the balance of resource used and achieve the constraint δ.

3.3 EXECUTION PLATFORM

The kern of our architecture is a Virtex-II Pro fabric, which can be partially reconfigured at run-time and provides additionally a hardcore embedded processor. In Figure 2(a) we show the embedded system architecture in more details. The reconfigurable part of the FPGA is divided in n slots. Each slot provides a OS service framework (Figure 2(b)). The local memory is used to support the communication between local components and the global shared memory is used to perform the communication with components running on the CPU. The local controller is used to manage the access to the local memory and the global controller, which together with its counterpart in software, performs the communication infrastructure mentioned in Section 3. The slots are connected using *Busmacros*. In order to program the FPGA slots, the reconfiguration port is used, which may be local (by using the ICAP Xilinx entity) or an external Run-Time Reconfiguration (RTR) controller.

4. COMMUNICATION-AWARE ALLOCATION ALGORITHM

The new algorithm is build on top of the already available allocation heuristic shortly described in Section 3.2. The idea is to group those components together who present lower communication costs when located at same execution domain. After this clustering process, we do apply the allocation algorithm, where not only single components are assigned to CPU or FPGA, but also meta-components (cluster of components). The previous algorithm is slightly modified in its second phase, when the components assignments are refined, in order to avoid the grouping of made at first.

4.1 DEFINITIONS AND NOTATIONS

The new allocation algorithm is based on component clustering. Therefore, we model our system as an undirected weighted graph $G = (\mathcal{V}, \mathcal{E})$. The edges in $\mathcal{E}$ represent the communication costs CM between two different components. Note that CM depends on two main factors: a static one, related with the architecture (time to deliver a message) and a factor related to the amount of data changed between two components, which is dynamic and depends on the application. Additionally, as each component may be located in one of two different execution environments, the communication cost CM performed between two components is noted by three different values $CM = \{C_\alpha,\ C_\beta,\ C_\gamma\}$:

- C_α, when both are on SW domain;
- C_β, when both are on HW domain;
- C_γ, when both are in different domains.

The Figure 3 shows a sample of such a graph. The grey nodes in the graph may be seen as the OS services primitives (API) that are made available for the application tasks (running in software). Note that such node do not have allocation costs as they only is used to properly represent the communication cost between an application task and an OS service.

To measure the connection degree between two communicating components, we define the *local preference* metric, $pl = \frac{2C_\gamma}{2C_\gamma + C_\alpha + C_\beta}$, which is calculated using CM ($pl = f(C_\alpha, C_\beta, C_\gamma)$). The metric pl compare the communication cost between two components when both are placed in the same execution domain in comparison with the case where each of them are placed in different execution domains.

We also define a *global preference* metric, pg, which is pl multiplied by a global factor (see Equation 4). This metric enable us to compare all local preferences with each other by doing the clustering process.

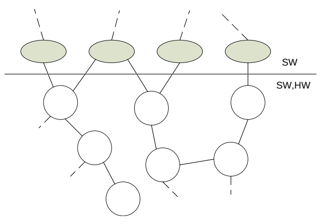


Figure 3. Sample of an OS component graph.

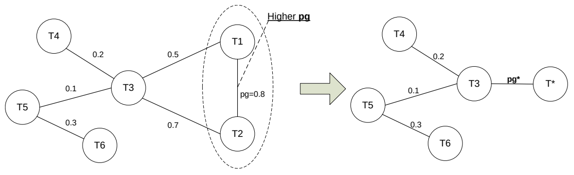


Figure 4. Example of a two component being clustered.

$$pg = (\frac{C_\gamma}{\max_V \{C_\gamma\}})pl \tag{4}$$

4.2 CLUSTERING COMPONENTS

The proposed clustering algorithm starts searching for the biggest global preference value, pg', among all edges and tries to cluster the two related components, o and p, respecting two conditions:

- Components o and p have not been clustered;
- $(c_{o,1} + c_{p,1},\ c_{o,2} + c_{p,2}) \leq (\lambda_1,\ \lambda_2)$, where λ_1 and λ_2 are the maximum component costs allowed when performing the combination of o and p. This criterium is used to avoid the deprecation of the allocation algorithm when the allocation costs of the formed components increase.

If a cluster is formed, the two involved components are combined and the search is executed again. This method is repeated until no more components are free for clustering.

When two components are grouped together, a new one is generated T^*. The Figure 4 shows an example. For this case, the CM^* will be generated as follows: $CM^* = CM_{1,3} + CM_{2,3}$. Thus, pg^* is calculated using this new value CM^*: $pg^* = f(C^*_\alpha,\ C^*_\beta,\ C^*_\gamma)$. Note that the communication costs, C_α and C_β, between $T1$ and $T2$ (from the example) are no longer considered for the pg evaluation. Nevertheless, they are stored and used during the balance improvement executed in second phase of the original algorithm ([5]).

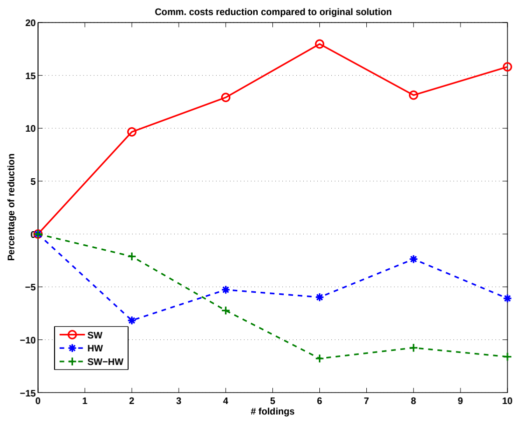


Figure 5. Evaluation results comparison.

4.3 COMPONENT ALLOCATION

When the clustering process ends, we apply the allocation algorithm, which is basically the same one presented in [5]. However, in the second phase of this algorithm, where the allocation decision is refined in order to fulfil the δ constraint (balance improvement), we use the stored C_α and C_β costs. Thus, a pair of components is allowed to change their location, only if this change will improve the balancing and also represent a reduction of the communication costs inside the execution domain.

5. EVALUATION RESULTS

For the evaluation of our algorithm, we did implement it using the MATLAB tool. We create the same environment used in the evaluation of the original algorithm and compare the communication costs achieved by the allocation with and without the clustering process. For our case, we generate randomly the communication graphs of $n = 20$ components respecting the following relation: $C_\alpha < C_\beta < C_\gamma$, which corresponds what we have observed in our architecture. (The communication costs across execution domains are the most expensive ones). The Figure 5 presents the results of the evaluation in every execution domain and also between them. It indicates in percentage, the increase of communication costs for a case using the clustering process in relation to the case where cluster was not used. Therefore, a negative percentage value indicates that a reduction of the communication costs was achieved. The results were performed for different number of clustered (folding) formed.

6. CONCLUSIONS

The paper presents a communication-aware allocation algorithm that is used for mapping OS services (components) of a system to FPGAs or CPUs. We give an idea how to evaluate the communication between the tasks and show how to cluster the tasks. This work is based on the approach presented in [5] and its extension is right now under development, but will be ready for the final version of the paper if it will be accepted.

REFERENCES

[1] Krishnamoorthy Baskaran, Wu Jigang, and Thamipillai Srikanthan. Hardware partitioning algorithm for reconfigurable operating system in embedded systems. In *Sixth Real-Time Linux Workshop*, November 2004. Singapore.

[2] Petru Eles, Krzysztof Kuchcinski, and Zebo Peng. *System Synthesis with VHDL: A Transformational Approach*, chapter 4, pages 114–119. Kluwer Academic Publishers, 1998.

[3] Frank Engel, Ihor Kuz, Stefan M. Petters, and Sergio Ruocco. Operating Systems on SoCs: A Good Idea? In *ERTSI Workshop*, 2004.

[4] Marcelo Götz and Florian Dittmann. Scheduling Reconfiguration Activities of Run-time Reconfigurable RTOS Using an Aperiodic Task Server. In *Proc. of the ARC 2006*, Delft, The Netherlands, March 2006.

[5] Marcelo Götz, Achim Rettberg, and Carlos E. Pereira. Towards Run-time Partitioning of a Real Time Operating System for Reconfigurable Systems on Chip. In *Proc. of IESS*, Manaus, Brazil, August 2005.

[6] J. Harkin, T. M. McGinnity, and L. P. Maguire. Modeling and optimizing run-time reconfiguration using evolutionary computation. *Trans. Embedded Comp. Sys.*, 3(4), 2004.

[7] Paul Kohout, Brinda Ganesh, and Bruce Jacob. Hardware support for real-time operating systems. In *International Symposium on Systems Synthesis*. Proc. of the 1st IEEE/ACM/IFIP Inter. Conf. on HW/SW codesign and system synthesis, 2003.

[8] P. Kuacharoen, M. Shalan, and V. Mooney. A configurable hardware scheduler for real-time systems. In *ERSA*, pages 96–101, June 2003.

[9] J. Lee, K. Ingström, A. Daleby, Tommy Klevin, V.J. Mooney III, and Lennart Lindh. A comparison of the rtu hardware rtos with a hardware/software rtos. In *ASP-DAC*, January 2003.

[10] Jaehwan Lee, Kyeong Ryu, and V. J. Mooney III. A framework for automatic generation of configuration files for a custom hardware/software rtos. In *ERSA*, June 2002.

[11] Lennart Lindh and Frank Stanischewski. Fastchart - a fast time deterministic cpu and hardware based real-time-kernel. In *EUROMICRO*, 1991.

[12] Jean-Yves Mignolet, Vincent Nollet, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *DATE*, pages 10986–10993, 2003.

[13] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable soc. In *IPDPS*, Washington, DC, USA, 2003. IEEE Computer Society.

[14] Grant Wigley and David Kearney. The development of an operating system for reconfigurable computing. In *FCCM*, pages 249–250, April 2001.