Reconfigurable Microkernel-based RTOS: Mechanisms and Methods for Run-Time Reconfiguration *

Marcelo Götz and Florian Dittmann
Heinz Nixdorf Institute
University of Paderborn
Fuerstenallee 11, 33102 Paderborn, Germany
{mgoetz, roichen}@upb.de

Abstract

The requirements of high computational performance and flexibility of the contemporary embedded systems are continuously increasing. Moreover, a single architecture must be able to support different applications with dynamical requirements (changing environments). Reconfigurable computing based on hybrid architectures, comprising general purpose processor (CPU) and Field Programmable Gate Array (FPGA), is very attractive because it can provide high computational performance as well as flexibility to support the requirements of today's embedded systems. An operating system (OS), which is desired to provide support for such systems, has to use the available resources in an optimal way (competing with the applications), since embedded system architectures are usually lacking in resources. In this paper, we present our approach towards a reconfigurable RTOS that is able to distribute itself over a hybrid architecture (comprising FPGA and CPU). We will describe the main concepts and methods used to achieve the desired RTOS. Moreover, we present some preliminary evaluation results which show the realizability of our approach.

1. Introduction

Operating Systems (OS) ease the application design and help to properly tackle the underlying architecture of a system [5]. Evaluating the reasons for using an OS regarding Systems-on-Chip (SoC), we find strong support for applying an OS to SoCs. However, the embedded system nature of SoCs is usually characterized by a limited availability of resources. Thus, a complete OS that supports all occurring application requirements is usually difficult to instantiate. Moreover, due to changing application requirements, an efficient usage of the available resources (shared between OS and application) is necessary. Here, reconfigurable computing comes in and provides a sophisticated architecture for OS requirements of a SoC.

Run-time reconfigurable architectures are becoming very attractive to compose run-time platforms for today's embedded systems. For instance, in modern Field Programmable Gate Arrays (FPGAs), the availability of a general purpose processor (GPP) surrounded by a large field of reconfigurable hardware offers the possibility for a sophisticated SoC concept. Moreover, the capability of such devices to be on-the-fly partially reprogrammed allows to dynamically adapt not only the software but also the hardware to the current system requirements, performing a Reconfigurable SoC (RSoC). The resulting system is one that can provide high performance by implementing custom hardware functions in the FPGA and still be flexible by reprogramming the FPGA and/or using a microprocessor (hybrid architecture).

Embedded systems often need to cope with different kinds of application, which may enter or leave the system over time. Additionally, each application may have different resource requirements during its operation. The Figure 1 shows such a scenario. For instance, imagine a Personal Data Assistant (PDA) where a movie is being played at full screen. Specific services are required from the application to achieve a predefined Quality-of-Service (QoS) (e. g., memory bandwidth to read the movie data, decoder performance, driver for display device). The user may, in an arbitrary point of time, open a new application (e. g., an e-mail client) and also choose to keep the movie running in a smaller screen. In this situation, the movie application will run in a different mode having, therefore, different

*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. It was also partially supported by DFG SPP 1148.

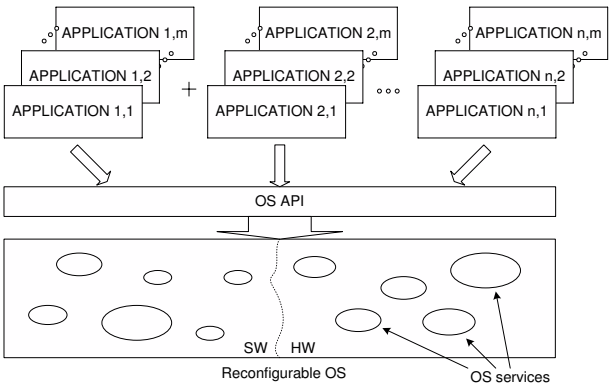


Figure 1. System overview

requirements (e. g., lower bit-rate decoder, small bandwidth memory access). Moreover, new resources may be provided to the new application.

In the scope of our ongoing research we are developing a run-time reconfigurable Real-Time Operating System (RTOS). We propose to reconfigure our OS online to provide the necessary services for the current application needs. Therefore, the system continuously analyzes the application requirements and decides on which execution domain (CPU or FPGA) the required RTOS components will be placed. Additionally, whenever the OS components are not placed in an optimal way (due to the application dynamics), a reconfiguration of the OS is necessary. Thus, techniques for a deterministic system reconfiguration need to be used in order to avoid the violation of the timeliness of the real-time running applications.

In this paper we will present the main aspects of our OS, focusing on the mechanisms used to handle the OS service allocation and reconfiguration at run-time. The remaining of the paper is organized as follows. After the analysis of the state-of-the-art in Section 2, we give an overview of the whole system in Section 3. The Section 4 gives the necessary model and notations used in Sections 5 and 6, which discuss the service allocation and the service reconfiguration, respectively. We provide the results of our experiments in Section 7 and give the conclusions in Section 8.

2. Related Work

Implementing OS services in hardware is a well researched field. Several works in the literature, e. g., [12], [11], [10], [13] and [14], show that hardware implementation may significantly improve performance and determinism of RTOS functionalities. The overhead imposed by the operating system, which is carefully considered in embedded systems design due to usual lack of resources, is considerably decreased by having RTOS services implemented in hardware. However, up to now all approaches have been based on implementations that are static in nature, that means, they do not change at run-time even when application requirements may change significantly.

Reconfigurable hardware/software based architectures are very attractive for implementation of run-time reconfigurable embedded systems. The hardware/software allocation of system components to dynamically reconfigurable embedded systems allows for customization of their resources during run-time to meet the demands of executing applications, as can be seen in [9].

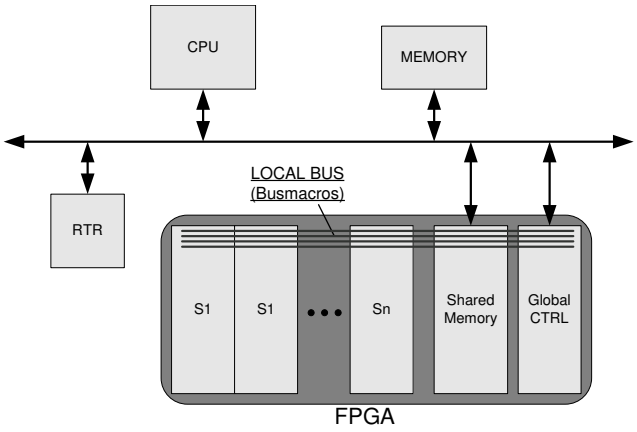
An example of this trend is the Operating System for Reconfigurable Systems (OS4RS) [16]. This work proposes an operating system for a heterogeneous RSoC. It aims to provide an on-the-fly reallocation of specific application tasks over a hybrid architecture, depending on the computational requirements and on the QoS expected from the application. Nevertheless, the RTOS itself is still static. Moreover, the time needed for reconfiguration is barely an issue in the design.

Additional research efforts spent in the field of reconfigurable computing are only focusing on the application level, leaving to the RTOS the responsibility to provide the necessary mechanisms and run-time support. The works presented in [17] and [15] are some examples of RTOS services to support the (re)location, scheduling and placement of application tasks on an architecture composed by an FPGA with or without a CPU. In our proposal, we expand those concepts and propose new ones to be applied on the RTOS level. Thus, not only the application but also the RTOS itself may be reconfigured on a hybrid architecture in order to make a better usage of the available resources in a flexible manner. Moreover, according to our knowledge there is very little work dealing with on-line migration of processes between hardware and software execution environments.

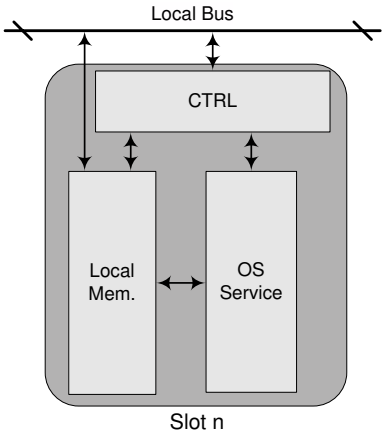
3. System Overview

3.1. Hardware Architecture

The core 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 into n slots. Each slot provides an 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 Sec-



(a) Architecture overviwe



(b) OS service slot template

Figure 2. System architecture

tion 3.2. 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.

3.2. Abstract View

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 versions of implementation: 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 3 shows our architecture abstractly. Additionally, the communication infrastructure layer provides the necessary support to

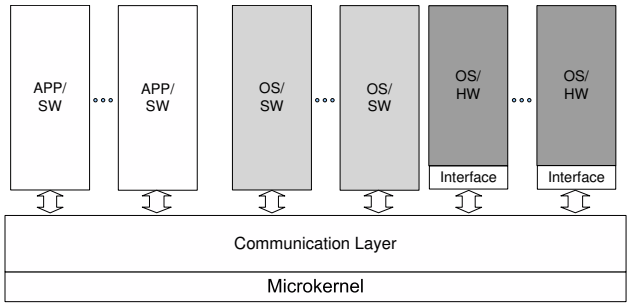


Figure 3. Proposed microkernel based architecture.

allow the communication among components running over the hybrid architecture in an efficient manner.

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

4. Model and Problem Formulation

The problem of assigning RTOS components to the two execution environments can be seen as a typical assignment problem. Therefore, we decided to model the problem using Binary Integer Programming (BIP) [18]. A set of available services is represented as $S = \{s_{i,j}\}$, where every service i has its implementation for CPU ($j = 1$) or FPGA ($j = 2$). Every component has an estimated cost $c_{i,j}$, which represents the percentage of resource from the execution environment used by this component. On the FPGA it represents the circuit area needed by the component, and on the CPU it represents the processor workload used by it.

The assignment of a component to either CPU or FPGA is represented by the variable $x_{i,j}$. We say that $x_{i,j} = 1$, if the component i is assigned to the execution environment j, and $x_{i,j} = 0$ otherwise.

Due to the application dynamism, the assignment decision needs to be checked continuously. This implies that a set of RTOS components needs to be relocated (reconfigured) by means of migration. In other words, a service may migrate from software to hardware or vice-versa. Additionally, services may be replaced by new ones (in order to use less/more resources). In this case, a reconfiguration of a service in the same execution environment occurs (hereafter also called migration).

In a typical embedded real-time system, the application may be modeled as a set of periodic activities. Sporadic tasks can also be modeled as periodic ones through the assumptions that their minimum interarrival time being the

Table 1. Service definition related to its periodic execution

Parameter	Description
$E_{sw,i}$	Execution time of a service i in software
$E_{hw,i}$	Execution time of a service i in hardware
P_i	Period of a service i
D_i, $d_{i,k}$	Relative and absolute deadlines of a service i
$a_{i,k}$, $s_{i,k}$, $f_{i,k}$	Arrival, starting and finishing time of a service i

period. Hence, we assume the applications as being a set of periodic tasks and that the OS services required present an periodic behavior.

Let us define a set $\mathcal{S}$ that represents the OS services: $\mathcal{S} = \{s_i,\ i = 1, ..., n\}$. Hereafter, we will use *component* and *service* as interchangeable terms. In relation to its periodic execution, each service $s_i \in \mathcal{S}$ is characterized by the parameters shown in Table 1, where k denotes the kth instance of a process i.

Additionally, every service i running in software utilizes some processor load which is defined as: $U_i = \frac{E_{sw,i}}{P_i} = c_{i,1}$. Similarly, for every service i running in hardware, some percentage of the FPGA area is used: $A_i = c_{i,2}$.

5. OS Service Allocation

A heuristic algorithm presented in [7] and improved in [8] determines the allocation OS components. 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 an objective cost function (Equation 1) subject to the system resource 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 environments. Thus, $\sum_{j=1}^{2} x_{i,j} = 1$ for every $i = 1, ..., n$. The weights w_1 and w_2 are used to properly compare the resource utilization between two different execution environments. The solution of this BIP are the assignment variables $x_{i,j}$, which we define as being a specific system configuration: $\Gamma = \{x_{i,j}\}$.

The allocation 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 [4]), 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 resources used and to achieve the constraint δ.

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 components needs to be relocated (reconfigured) by means of migration. In other words, a service may migrate from software to hardware or vice-versa.

6. OS Service Reconfiguration

As is has been said in the Section 4, the application requirements are considered to change over system life time. These modifications are represented by changes of the component costs $c_{i,j}$. This leads to the fact that a certain system configuration Γ may no longer be valid after application changes. Therefore, a continuously evaluation of the components partitioning is necessary. Whenever the systems reaches an unbalanced situation ($|w_1U - w_2A| > \delta$), the RTOS components should be reallocated in order to bring the system again in the desired configuration.

In this case, a subset $\mathcal{SR}$ of the current active service set $\mathcal{S}$ will suffer a reconfiguration (hereafter called migration): $\mathcal{SR} = \{s_i^\star,\ i = 1, ..., m\}$, where $\mathcal{SR} \subseteq \mathcal{S}$ and $m \leq n$. From this point, we have to solve two problems:

- the order in which the components will be reconfigured;
- the reconfiguration from each component itself.

Both cases will be treated in the following subsections.

6.1. OS Components Scheduling

Defining $\mathcal{TS}$ and $\mathcal{TH}$ as the services running in software and hardware, respectively, after one migration we will have: $\mathcal{TS}^*$ and $\mathcal{TH}^*$. Thus, if for every component migration the following conditions are fulfilled, the schedulability of the tasks are guaranteed:

$$\sum_{i \in \mathcal{TS}^*} U_i \leq U_{max} \; ; \; \sum_{i \in \mathcal{TH}^*} A_i \leq A_{max} \qquad (4)$$

In order to find a feasible schedule for every task migration, the service subset $\mathcal{SR}$ (defined in Section 6) needs to be previously sorted in a proper order. In other words, the sorting of services that will suffer a reconfiguration characterizes a schedule problem under resources constraints.

In order to tackle this problem in a proper manner, we separate the components that will undergo a migration in their own execution environment from the ones that will change it. Given the set $\mathcal{SR}$, three new subsets are defined:

$\mathcal{SR}^a$ Services that will be migrated in software;

$\mathcal{SR}^b$ Services that will be migrated in hardware;

$\mathcal{SR}^c$ Services that will be migrated between hardware and software.

The services from subset $\mathcal{SR}^a$ will be scheduled first if they will represent a reduction in the final CPU workload, otherwise, they will be scheduled at the end. The same is done with the components from $\mathcal{SR}^b$ concerning the final FPGA area. Therefore, our proposed approach is reduced by finding a schedule for the service set $\mathcal{SR}^c$.

If $|\mathcal{SR}^c| = x$, the feasible solutions S_f is a subset of the $x!$ possible schedule solutions (permutations of components in $\mathcal{SR}^c$). To solve this problem, Bratley's algorithm [1] could be applied, in which the search space for a valid schedule is reduced. Nevertheless, the worst-case complexity of the algorithm is still $O(x \cdot x!)$, as we have to analyze $x!$ paths of length x. For this reason, we propose a heuristic algorithm to solve the component reconfiguration schedule.

The basic idea of our heuristic algorithm is the use of the component costs ($c_{i,j}$) as a criteria for searching a solution in the tree of all possible schedules. Looking at the components that need to leave the CPU, the strategy is the following: try to migrate the component with the highest software cost and with the smallest hardware cost first. Thus, the total software resources used tend to decrease quickly and, in the same way, the total hardware resources used tend to increase slowly. Similarly, the same strategy is applied to the components that need to leave the FPGA. Consequently, two partial schedules PSa and PSb are generated using the strategy explained above.

Let $Sa = \{sa_1, ..., sa_p\}$ and $Sb = \{sb_1, ..., sb_q\}$ be the components that need to leave the CPU and FPGA, respectively, so that $Sa \cup Sb = \mathcal{SR}^c$ and $Sa \cap Sb = \emptyset$. Let $Ia = \{i_1, ..., i_p\}$ be the index array that represents Sa sorted by decreasing software costs, so that $\{c_{i_1,1} \geq c_{i_2,1} \geq ... \geq c_{i_p,1}\}$. Similarly, $Ja = \{j_1, ..., j_p\}$ is defined as the index array that represents Sa sorted by increasing hardware costs: $\{c_{j_1,2} \leq c_{j_2,2} \leq ... \leq c_{j_p,2}\}$.

The algorithm starts comparing the first two components of Ia and Ja ($k = 1$). If no match (same index in both arrays) is found, it expands the search ($k = 2$) on the first two components of Ia and Ja (total of four components). Hence, the search is done gradually until a match is found. If this is the case, the index is removed from both arrays, the schedule is updated and the search restarts on the remaining arrays. Note that a match is always found, since the same elements from Ia is also presented in Ib. Hence, the algorithm will always terminate.

It can be seen that for every k value the algorithm calculates, in the worst-case, $2k - 1$ comparisons. Thus, for a worst-case scenario when searching for a match, where the search is done over the whole array ($k = |Ia| = |Sa|$), the total number of comparisons will be $1 + 3 + 5 + ... + (2p - 1) = \sum_{i=1}^{p}(2i - 1) = p^2$ (which is the maximum number of combinations that can be done between two arrays of size p). Therefore, the complete partial schedule algorithm has a complexity of $O((n - 1)n^2)$, since for every index match found, the search will be applied again on a reduced index array.

If Ib and Jb are defined as the index arrays that represent Sb sorted by decreasing hardware costs and sorted by increasing software costs, respectively, we apply the same partial algorithm using these two index arrays to get PSb.

The final schedule is found by merging the partial schedules in an interleaving manner. The components from the partial schedules PSa and PSb are selected in an alternating manner. The number of components selected from each partial schedule at each step is proportional to their sizes ($|PSa|$ and $|PSb|$).

6.2. OS Component Reconfiguration

To handle the reconfiguration of each single OS component in a deterministic way, we propose to model these reconfiguration activities as aperiodic jobs and therefore a server for aperiodic jobs is applied. As the arrival of these reconfiguration activities is not known a priori, they can be seen as aperiodic jobs.

Let us represent these reconfiguration activities as a set $\mathcal{J}$ of aperiodic jobs: $\mathcal{J} = \{J_i(J_i^a, J_i^b),\ i = 1, ..., m\}$. In real-time scheduling theory, when real-time periodic tasks and non (or firm) real-time aperiodic tasks need to be scheduled together, a server for aperiodic tasks is generally used. The basic idea of this approach is to include a new periodic task into the system, which will be responsible for carrying out the aperiodic jobs without causing a periodic task to miss its deadline. A more comprehensive and detailed explanation of this idea is given in [3].

Among different types of servers, we focus our analysis on the Total Bandwidth Server (TBS) [3] due to the following reasons:

- We are currently using Earliest Deadline First (EDF) as our schedule policy;
- Under EDF, it is one of the most efficient service mechanism in terms of performance/cost ratio [2].

According to the literature, the TBS assigns a deadline for an aperiodic task k arriving in the system at time a_k in the following manner:

$$d_k = \max(a_k, d_{k-1}) + \frac{C_k}{U_s} \tag{5}$$

Where d_{k-1} represents the deadline of the aperiodic job that has arrived before job k; U_s is the server bandwidth and C_k is the execution time requested by the aperiodic job. Deadline d_{k-1} is 0 if k is the first one, or if all pending aperiodic jobs have arrived before k has already been finished.

Since we consider the behavior of a service execution as being periodic (see Table 1), we constraint the migration activity (the periodic job) to be carried out between two consecutive instances of a service. Thereby, we avoid the preemption of a service during its execution in one execution domain and the resume of this service in the other execution domain. Moreover, we reduce significantly the amount of context data migration between execution domains. In order to clarify this situation, in Figure 4 a scenario is presented where an OS service is migrated from software to hardware.

Assuming this constraint, we adjust the arrival of a migration job to the arrival of a service instance and the deadline for this job to the beginning of the service execution (start time). These adjustments and the execution time of a migration job are then applied in the Equation 5 (TBS deadline assignment rule). Hence, we now can derive the minimal server bandwidth necessary to migrate an OS service respecting the constraint explained above. For each possible migration case a minimal server bandwidth was derived and the results can be seen in [6] and will not be treated in the scope of this work.

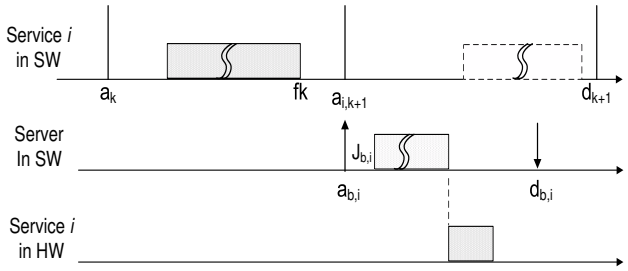


Figure 4. Scenario where a service migrates from software to hardware

7. Experimental Results

For system evaluation of the run-time assignment problem, we made some simulations using the MATLAB tool. We generated a number of 100 different systems having randomly costs: $1\% \leq c_{i,1} \leq 15\%$ and $5\% \leq c_{i,2} \leq 25\%$; and fixed size: $n = 20$ components. The maximum FPGA resource was defined to be 100% ($A_{max} = 100$), as well as for the CPU ($U_{max} = 100$). The components assignment were calculated for every system using the 0-1 Integer Programming (optimal solution) and the heuristic algorithm (first and second one). The average value of total cost ($U + A$) and the absolute difference cost ($|w_1 U - w_2 A|$) were compared for different values of δ (the resource usage balancing restriction): $(0.5, 1, 2, 3, 4, 5, 10, 20, 30, 40, 50\ and\ 60)$.

The solutions provided by the first heuristic algorithm were very similar to the optimal one, if the δ constraint has values around ($\delta \approx 10\%$), concerning the fulfillment of this constraint (see Figure 5, Heuristic-1). The smaller the δ the poor the results given by the first heuristic algorithm. This was expected to be so, since the first algorithm does not consider the balancing restriction.

The application of the second algorithm over the solution provided by the first one delivers a better balancing B. However, an increase in the total cost assignment was verified for the cases where the second algorithm achieved an improvement in the balancing B ($\delta <\approx 10\%$). Nevertheless, the total cost assignment achieved by this heuristic algorithm were quite satisfactory: maximum of 15% bigger if compared with the optimal case (see Figure 6, Heuristic-2).

The scheduling of OS components was also evaluated. Therefore, two consecutive system generated randomly for the previous analysis was considered to be a system pair to representing the current and new system configuration. To evaluate the efficiency of the heuristic algorithm, all possible feasible solutions were also generated by calculating all possible permutations of the reconfigurable components. Due to computation complexity restriction, the size of the

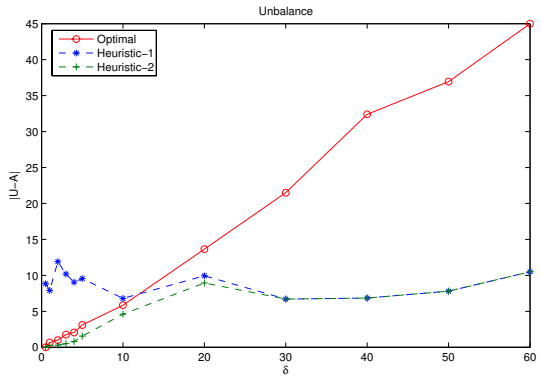


Figure 5. Unbalance average for different δ constraints.

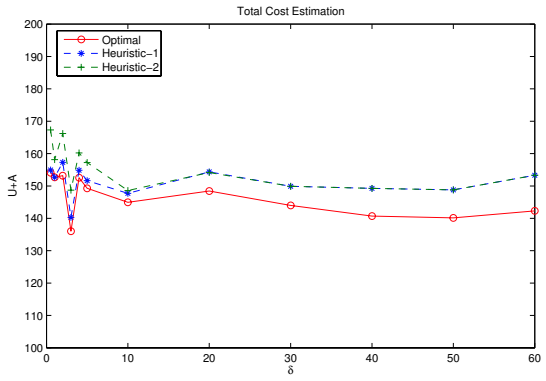


Figure 6. Total cost assignment average for different δ constraints.

systems generated was limited to 8, which may produce a maximum of 8! possible solutions.

Starting from $U = A = 100\%$ and systems having cost average of 10%, the system did have 99.9 times at least one feasible solution. From this amount of cases, the algorithm could find a solution in almost all cases: 98.9%.

The dashed line at Figure 7 shows the percentage of cases (average values) where at least one feasible solution was found. The solid line shows the percentage of cases where the heuristic did find a feasible solution from the feasible ones. From Figure 7 it can be seen that the smaller the U and A constraints became the poorer the efficiency of the heuristic algorithm was. Nevertheless, the efficiency of the heuristic algorithm decreases very much slower than the number of cases where any feasible solution exists.

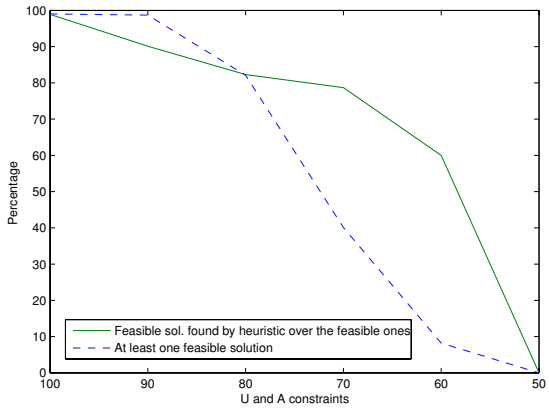


Figure 7. Heuristic Algorithm Evaluation

8. Conclusions and Future Work

Based on the need for OSs for SoCs and the contemporary capabilities of reconfigurable devices, we presented our concepts and methods used to build a run-time reconfigurable RTOS for SoC in this paper. The RTOS is able to reconfigure itself over a hybrid architecture comprising a CPU and a FPGA. Additionally to the allocation and migration concepts used, we also presented the hardware architecture on which the RTOS and the applications run.

The RTOS is component based, where each component represents a service used by the application. As fundamental requirement for RTOS reconfiguration, we presented and evaluated a heuristic algorithm, which is used to allocate the OS components over the hybrid architecture. Additionally, we proposed the application of a server technique (from real-time scheduling theory) to schedule the reconfiguration (migration) activities in a deterministic manner. Therefore, we also proposed a heuristic algorithm to sort the reconfiguration activities in a proper order.

Currently, we are porting some specific services of an OS (e. g., encryption and a stack protocol of a communication channel) to our architecture. We plan to run and evaluate some prototyping scenarios on the architecture. Moreover, techniques for a precisely measurement of the components costs at run-time are being developed.

References

[1] P. Bratley, M. Florian, and P. Robillard. Scheduling with Earliest Start and Due Date Constraints. *Naval Research Logistics Quarterly*, pages 511–519, December 1971.

[2] G. C. Buttazzo. Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems*, 29(1):5–26, 2005.

[3] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, Second Edition*. Springer, 2006.

[4] P. Eles, K. Kuchcinski, and Z. Peng. *System Synthesis with VHDL: A Transformational Approach*, chapter 4, pages 114–119. Kluwer Academic Publishers, 1998.

[5] F. Engel, I. Kuz, S. M. Petters, and S. Ruocco. Operating Systems on SoCs: A Good Idea? In *Embedded Real-Time Systems Implementation (ERTSI) Workshop*, Lisbon, Porgutal, December 2004.

[6] M. Götz, F. Dittmann, and C. E. Pereira. Deterministic Mechanism for Run-time Reconfiguration Activities in an RTOS. In *of the 4th International IEEE Conference on Industrial Informatics - INDIN*, Singapore, 2006.

[7] M. Götz, A. Rettberg, and C. E. Pereira. Towards Run-time Partitioning of a Real Time Operating System for Reconfigurable Systems on Chip. In *Proceedings of International Embedded Systems Symposium 2005 - IESS*, Manaus, Brazil, August 2005.

[8] M. Götz, A. Rettberg, and C. E. Pereira. Communication-Aware Component Allocation Algorithm for a Hybrid Architecture. In *Proceedings of 5th IFIP Working Conference on Distributed and Parallel Embedded Systems - DIPES 2006*, October 2006.

[9] J. Harkin, T. M. McGinnity, and L. P. Maguire. Modeling and optimizing run-time reconfiguration using evolutionary computation. *Transactions on Embedded Computing Systems*, 3(4):661–685, 2004.

[10] P. Kohout, B. Ganesh, and B. Jacob. Hardware Support for Real-time Operating Systems. In *International Symposium on Systems Synthesis*, pages 45–51, 2003.

[11] P. Kuacharoen, M. Shalan, and V. Mooney. A Configurable Hardware Scheduler for Real-Time Systems. In *ERSA*, June 2003.

[12] J. Lee, V. J. M. III, K. Ingstrm, A. Daleby, T. Klevin, and L. Lindh. A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS. In *ASP-DAC2003*, page 6, 2003. Japan.

[13] J. Lee, K. Ryu, and V. J. M. III. A Framework for Automatic Generation of Configuration Files for a Custom Hardware/Software RTOS. In *ERSA*, June 2002.

[14] L. Lindh and F. Stanischewski. FASTCHART - A Fast Time Deterministic CPU and Hardware Based Real-Time-Kernel. In *EUROMICRO'91*, pages 12–19, 1991. Paris, France.

[15] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip. In *DATE*, 2003.

[16] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In *International Symposium on Parallel and Distributed Processing - IPDPS*. IEEE Computer Society, 2003.

[17] G. Wigley and D. Kearney. The Development of an Operating System for Reconfigurable Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM*, pages 249–250, April 2001.

[18] L. A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.