Deterministic Mechanism for Run-time Reconfiguration Activities in an RTOS

Marcelo Götz
Heinz Nixdorf Institute
University of Paderborn, Germany
mgoetz@upb.de

Florian Dittmann
Heinz Nixdorf Institute
University of Paderborn, Germany
roichen@upb.de

Carlos Eduardo Pereira
Electrical Engineering Department
UFRGS, Brazil
cpereira@ece.ufrgs.br

***Abstract*— 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. However, the relative high reconfiguration costs often are an obstacle when using such architectures for run-time reconfigurable systems. In order to still be able to benefit from the flexibility of such systems, the used real-time operating system must explicitly respect the reconfiguration time. In such systems, the reconfiguration activities need to be carried out during run-time without causing critical applications to miss their deadlines. In this paper, we show how we model these reconfiguration activities as aperiodic jobs. Therefore, we apply the server-based method from the real-time scheduling theory to the scheduling of aperiodic activities. Using these techniques, we can achieve a deterministic environment for reconfiguration activities as well as an improvement of their responsiveness.**

I. INTRODUCTION

Run-time reconfigurable architectures are becoming very attractive to compose run-time platforms for today's embedded systems. For instance, in currently available 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 System-on-Chip (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 kind of applications, which may enter or leave the system over time. Additionally, each application may have different resource requirements during its operation. For instance, just 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 requirements (e.g., lower bit-rate decoder, small bandwidth memory access). Moreover, new resources may be provided to the new application.

The usage of an Operating System (OS) will ease the application design and help to properly tackle the underlying architecture. Actually, the reasons for using an OS for SoC are similar to those for running an OS on any system [1]. However, due to the limited availability of resources, a complete OS that supports all applications is usually difficult to instantiate. Moreover, due to the changing application requirements, an efficient usage of the available resources (shared between OS and application) is necessary.

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 dynamism), 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 the scope of this paper, we will focus on the mechanism used to handle the reconfiguration activities in a deterministic way applying techniques from real-time scheduling theory. Our proposal consists of modeling these reconfiguration activities as aperiodic jobs and therefore a server for aperiodic jobs is applied.

The remainder of the paper is organized as follows. After summarizing related work, we introduce the execution platform. In order to schedule the reconfiguration activities, we first derive the model and definitions used. In Section V, we explain the appliance of a server in our scenario. The schedulability analysis is presented in Section VI and a heuristic algorithm is proposed and tested for sorting the components for reconfiguration and thus guaranteeing the global schedulability in Section VII. We also analyze the performance of our algorithms by doing some simulations

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.

1-4244-9701-0/06/$20.00

in Section VIII. Finally, we conclude and give an outlook in Section IX.

II. RELATED WORK

The idea of implementing OS services in hardware is not new. Several works in the literature, [2], [3], [4], [5] and [6], 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 [7].

An example of this trend is the Operating System for Reconfigurable Systems (OS4RS) [8]. 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 not a big 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 [9] and [10] 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 are just few works dealing with on-line migration of processes between hardware and software execution environments.

III. EXECUTION PLATFORM

Our target architecture is composed of one CPU, an FPGA, memory and a bus connecting the components. Most of these elements are provided on a single die, such as the Virtex II Pro, from Xilinx company. The RTOS services that are capable of being reconfigured are stored on an external SDRAM chip in two different implementations: as software object and as FPGA configuration bitstreams. Additionally, to program the FPGA, there is a Run-Time reconfiguration Manager (RTM) available that is able to run in parallel with hardware and software tasks (thanks to the partial reconfiguration capability of the FPGA used).

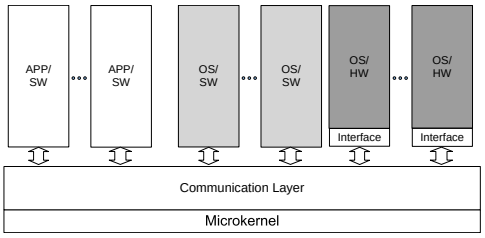

Fig. 1. Proposed microkernel based architecture.

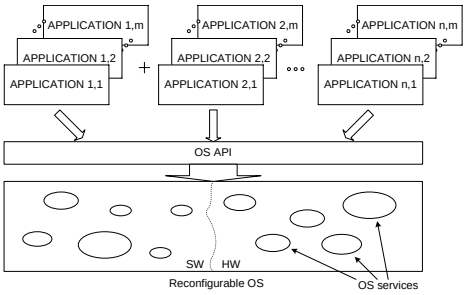

Fig. 2. System overview

Abstractly, the architecture can be seen as represented in Figure 1. The system concept is based on the microkernel approach. As a target RTOS, we are using DREAMS [11], whose components can be reallocated (reconfigured) over the hybrid architecture at run-time. The usage of a microkernel also incorporates the natural advantage of flexibility and extensibility (among others), which is inevitable in our case in order to better perform the reconfigurability aspects. The Communication Layer presented in Figure 1 provides an efficient communication infrastructure. It offers a transparent set of services to the application, independent of their locations.

IV. PROBLEM SPECIFICATION AND MODELING

As explained before, an application may have different resource requirements during its life time. Additionally, more than one of such applications may be running on the system. This scenario can be represented as shown in Figure 2. From the point of view of the OS, different set of services are required at different points of time, depending on the current application scenario. As the OS services are represented by components, each one (i) is assigned a specific cost $c_{i,j}$. It represents the percentage of CPU used if located in software ($j = 1$) or the percentage of FPGA area used if located in hardware ($j = 2$). Hence, their allocation over the hybrid architecture is done respecting the limited FPGA area (A_{max}) and CPU workload (U_{max}). The results of our investigation to solve this problem can be seen in [12] and [13].

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-

TABLE I

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

TABLE II

TIME COSTS RELATED TO EACH MIGRATION CASE

Parameter	Description
$RP_{sw,i}$	Programming time in software
$RP_{hw,i}$	Programming time in hardware
$RM_{sw,i}$	Migration time from software to software
$RM_{hw,i}$	Migration time from hardware to hardware
$RM_{shw,i}$	Migration time between hardware/software

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 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 I, 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: $A_i = c_{i,2}$.

The activity to reconfigure a component is divided in two distinct phases called Programming and Migration and denoted by RP and RM, respectively. During RP phase, the component is programmed on the FPGA (by downloading the corresponding partial bitstream), or the object code is placed in the main memory (if its software version is going to be used). Note that by the end of RP, the service is still not active (available for the application).

Once the FPGA is programmed with the service's bitstream, or the CPU software is linked with the service's object code, the effective migration can start (RM phase). This phase comprises the activity of transferring the internal data of a service between two execution environments and its activation. Table II shows the notations used to represent the different migration costs.

Although there are some methods that allow the preemption of hardware tasks as *readback* and *scan-path* (e.g., [14]), we do not allow that a service may be preempted and resumed across execution domains. This would incorporate a prohibited overhead cost to the system due to the transfer time of the service context data. In order to promote the migration to occur between two consecutive instances of a service i, two constraints are defined:

- **C1** The phase RM_i cannot start if an instance of a service i has been started or if this instance has not been finished yet.
- **C2** Once a phase RM_i has been started, it cannot be preempted by the service i.

As some small amount of data may still be required to be transferred by migration (e.g., some service parameters), each component contains an individual set of registers (provided in both hardware and software versions) used to read/write these data. Based on our current implementation status, we are considering the RM time as being the same for a task migrating from software to hardware or vice-versa (parameter RM_{shw}).

V. HANDLING RECONFIGURATION ACTIVITIES

As discussed previously, the OS reconfiguration occurs by means of component reconfiguration due to the changing environment. 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 $m \leq n$. This will represent that, for every service, the respective RP and RM phases will need to be executed. 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: $\mathcal{J} = \{J_i(J_i^a, J_i^b),\ i = 1, ..., m\}$. The execution time of job J^a may be either RP_{sw} or RP_{hw}, depending on which direction the component i will be migrated. Similarly, J^b is one of the RM migration times: $RM_{sw,i}$, $RM_{hw,i}$ or $RM_{shw,i}$ (see Table II).

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 [15].

Among different types of servers, we focus our analysis on the Total Bandwidth Server (TBS) [15] 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 [16].

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

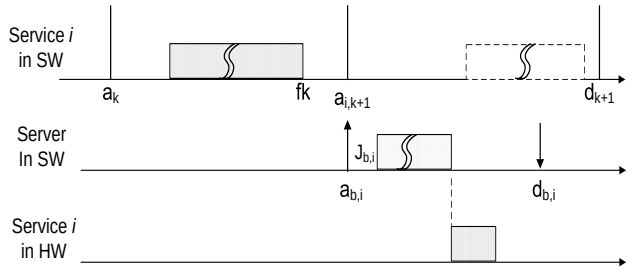

Fig. 3. Scenario where a service migrates from software to hardware

By using a server we consider the execution of the reconfiguration activities as being sequentially. This does not represent a degradation of the hardware parallelism since the availability of a single configuration port on today's FPGA serializes its programming operation.

When programming a service in software (RP_{sw}), the bandwidth of the server is spent to execute this activity. Similarly, when programming the partial bitstream onto the FPGAs (RP_{hw}), the server bandwidth is used. As during RP phase no extra synchronization with the running tasks or services are required, the appliance of the deadline assignment rule from TBS to schedule J^a is straightforward.

The job J_i^b, however, can just start after finishing the current instance of the service i and cannot be preempted by the new instance of the same service (constraints **C1** and **C2** are explained in Section IV). Therefore, in the next subsections we will analyze, for every migration case, the influence of **C1** and **C2** to the schedule of J_i^b.

A. Software to hardware migration

In order to facilitate the following analyses, let us define $a_{b,i}$ and $d_{b,i}$ as the arrival and deadline times of a job J_i^b. To ensure that J_i^b will not start before the instance $s_{i,k}$, we free the J_i^b execution only when $s_{i,k}$ has finished: $a_{b,i} \geq f_{i,k}$. In practice, it is possible to control the arrival time $a_{b,i}$ to be at the finishing time of $f_{i,k}$. Nevertheless, we need to analyze the worst case scenario in order to derive the minimal server bandwidth necessary to respect the constraints **C1** and **C2**. Figure 3 shows this situation where the arrival of the J_i^b occurs at the arrival time of the next service instance $a_{i,k+1}$. Note that we consider the relative deadline D_i as being equal to the period P_i.

Under EDF schedule, the running task is always the one with the smallest absolute deadline among the ready ones. Thus, we can ensure that J_i^b will not be preempted by $s_{i,k+1}$ making $d_{b,i} \leq d_{i,k+1}$. As the software component s_i will go to hardware, its CPU utilization factor U_i is released and can be added to the server bandwidth. Moreover, the J_i^b phase needs to be finished in the worst-case at $d_{i,k+1} - E_{hw,i}$ to allow the next instance of the service i (running in hardware) to finish before the deadline. Hence, the deadline $d_{b,i}$ assigned to J_i^b using TBS is:

$$d_{b,i} = a_{b,i} + \frac{RM_{shw,i}}{U_s + U_i} \leq d_{i,k+1} - E_{hw,i} \qquad (1)$$

Noting that $d_{i,k+1} = a_{i,k+1} + P_i$ and $a_{i,k+1} = a_{b,i}$, the Equation 1 can be rewritten in order to derive the minimal server bandwidth needed:

$$U_s \geq \frac{RM_{shw,i}}{P_i - E_{hw,i}} - U_i \qquad (2)$$

Equation 2 shows that the bandwidth required to migrate job $J_{b,i}$ is equal to $\frac{RM_{shw,i}}{P_i - E_{hw,i}}$. Hence, as the server uses the workload released by service i, U_s needs to be at least equal to the difference necessary to achieve the requested bandwidth.

If the condition expressed in Equation 2 is fulfilled under all services that will suffer a migration from software to hardware, we can guarantee that the conditions **C1** and **C2** will be satisfied when EDF and TBS are used.

B. Software to software migration

Applying a similar analysis as made above for the case where a service is going to be replaced by another service, but only in software domain (a software to software migration case), we need the following minimal server bandwidth:

$$U_s \geq \frac{RM_{sw,i}}{P_i} + U_i^{new} - U_i \qquad (3)$$

Where U_i^{new} is the new processor load used by service i after migration. Note that we do not change the period P_i, only the execution time: $E_{sh,i}^{new}$. It this case, the bandwidth requested by J_i^b is $\frac{RM_{sw,i}}{P_i}$ and the processor load released is $U_i - U_i^{new}$.

C. Hardware to software migration

For this migration case, the arrival time of job J_i^b can be defined more precisely since a service executing in hardware profits from the true parallelism of this environment. Additionally, the server needs to also provide the workload that will be reclaimed by service i when running in software. Hence, the remaining bandwidth $(U_s - U_i)$ needs to be big enough to migrate J_i^b. The deadline assigned to J_i^b by TBS is then defined as:

$$d_{b,i} = a_{i,k} + E_{hw,i} + \frac{RM_{shw,i}}{U_s - U_i} \leq d_{i,k+1} \qquad (4)$$

Noting that $a_{i,k+1} = a_{i,k} + P_i$ and $d_{i,k+1} = a_{i,k+1} + P_i$, we can rewrite the equation above in order to derive the minimal server bandwidth required:

$$U_s \geq \frac{RM_{shw,i}}{2P_i - E_{hw,i}} + U_i \qquad (5)$$

D. Hardware to hardware migration

For this case, an OS component is going to be reconfigured in the hardware execution domain only (a hardware to hardware migration case). Similar to the case above we have: $a_{b,i} = a_{i,k} + E_{hw,i}$. Moreover, the deadline to finish J_i^b can be precisely defined as: $d_{b,i} = d_{i,k+1} - E_{hw,i}^{new}$. Note that here also the period P_i is the same and only the execution time of the service i is different since it is going to be replaced. Thus, the assigned deadline to job J_i^b is:

$$a_{i,k} + E_{hw,i} + \frac{RM_{hw,i}}{U_s} \leq d_{i,k+1} - E_{hw,i}^{new} \qquad (6)$$

As the temporal distance between $a_{i,k}$ and $d_{i,k+a}$ is $2P_i$, the minimal server bandwidth for this case is:

$$U_s \geq \frac{RM_{hw,i}}{2P_i - E^{new}_{hw,i} - E_{hw,i}} \tag{7}$$

VI. SCHEDULABILITY ANALYSIS

All analyses made in the sections above were based on the proper assignment of arrival time and deadline of aperiodic reconfiguration activities and the establishment of conditions for the definition of the server bandwidth. These analyses were made in order to properly represent the precedence constraints imposed in our scenario (**C1** and **C2**). Thus, a schedulability analysis is still necessary.

Nevertheless, as we are restricting ourselves to a scenario were periodic tasks do not have dependencies, a simple schedulability analysis can be done. Additionally, in our approach we do not modify the deadline or arrival times of any periodic service. As a consequence, if after every migration step the sum of all processor utilizations (used by every software component) plus the server bandwidth does not exceed a maximum (U_{max}), the feasibility of the schedule is guaranteed.

As we also have a hardware constraint A_{max} (maximum FPGA area available), the FPGA needs to have the area requested by a task being located at this environment available.

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:

$$\sum_{i \in \mathcal{TS}^*} U_i \leq U_{max} \; ; \; \sum_{i \in \mathcal{TH}^*} A_i \leq A_{max} \tag{8}$$

In order to find a feasible schedule for every task migration, the service subset $\mathcal{SR}$ (defined in Section IV) 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.

VII. COMPONENT RECONFIGURATION SCHEDULING

In order to tackle this problem in a proper manner, we separate the components that will suffer a migration in its 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 [17] 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

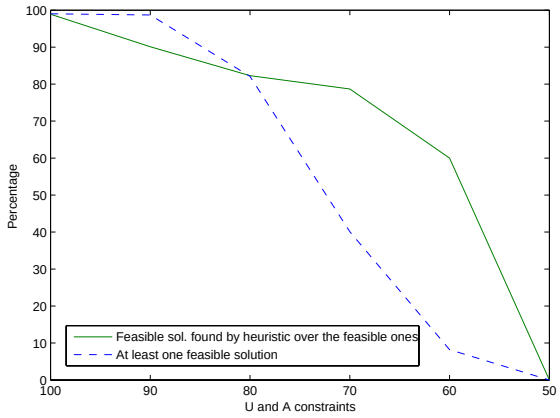

Fig. 4. Heuristic Algorithm Evaluation

from each partial schedule at each step is proportional to their sizes ($|PSa|$ and $|PSb|$).

VIII. SIMULATION RESULTS

In order to evaluate the efficiency of the proposed heuristic algorithm, random system pairs (representing the current and new system configuration) were randomly generated and the algorithm was applied to them. Every system pair generated was tested with different system constraints: $U = A = [100\%,\ 90\%,\ 80\%,\ 70\%,\ 60\%\ and\ 50\%]$. The component costs were randomly generated with averages going from 5% and 10%.

We repeated the evaluation 100 times. Moreover, the whole test was repeated 10 times, where in each case systems with different cost granularities were produced. Thereby, we tried to generate results independent from the system case. 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 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 4 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 4 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.

IX. CONCLUSION AND FUTURE WORK

In this work we have presented methods to handle the migration of RTOS components over a hybrid architecture in a deterministic manner by using a server. The concept proposed explicitly respects the reconfiguration time and imposes conditions for definition of the server bandwidth. By setting the server bandwidth properly, we can perform the migration deterministically without requiring to preempt and resume a service execution or its migration activity.

A heuristic that provides the scheduling of components to be reconfigured in our system has also been proposed. Its worst case complexity ($O(n^3)$) is significantly lower that of Bratley's algorithm usually used for such cases. In addition, we did show that our heuristic algorithm provides a good performance by doing some evaluations.

In the future we want to derive an analysis that will be able to determine the amount of time necessary to migrate/reconfigure the complete set $\mathcal{SR}$. Thus, an acceptance test for a reconfiguration case can also be derived.

REFERENCES

[1] 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.

[2] 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*, 2003, p. 6, Japan.

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

[4] P. Kohout, B. Ganesh, and B. Jacob, "Hardware Support for Real-time Operating Systems," in *International Symposium on Systems Synthesis*, 2003, pp. 45–51.

[5] 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.

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

[7] J. Harkin, T. M. McGinnity, and L. P. Maguire, "Modeling and optimizing run-time reconfiguration using evolutionary computation," *Transactions on Embedded Computing Systems*, vol. 3, no. 4, pp. 661–685, 2004.

[8] 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.

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

[10] 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.

[11] C. Ditze, "Towards Operating System Synthesis," Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Entwurf Paralleler Systeme, 2000, ISBN 3-931466-75-2.

[12] 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.

[13] ——, "Run-time Reconfigurable Real-Time Operating System for Hybrid Execution Platforms," in *of the 12th IFAC Symposium on Information Control Problems in Manufacturing - INCOM*, Sain-Etienne, France, 2006.

[14] A. Ahmadinia, C. Bobda, D. Koch, M. Majer, and J. Teich, "Task scheduling for heterogeneous reconfigurable computers," in *SBCCI*. New York, NY, USA: ACM Press, 2004, pp. 22–27.

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

[16] ——, "Rate Monotonic vs. EDF: Judgment Day." *Real-Time Systems*, vol. 29, no. 1, pp. 5–26, 2005.

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