A Run-time Partitioning Algorithm for RTOS on Reconfigurable Hardware *

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] Departamento de Engenharia Eletrica
UFRGS - Universidade Federal do Rio Grande do Sul - Brazil
cpereira@eletro.ufrgs.br

Abstract. In today's system design, reconfigurable computing plays more and more an important role. By the extension of reconfigurable devices like FPGAs with one or more CPUs new challenges in system design should be solved. These new hybrid FPGAs (e.g. Virtex-II Pro^{TM}), provides a hardcore general-purpose processor (GPP) embedded into a field of programmable gate arrays. Furthermore, they offer partial reconfiguration. Therefore, those hybrid FPGAs are very attractive for implementation of run-time reconfigurable embedded systems. However, most of the efforts in this field were made in order to apply these capabilities at application level, leaving to the Operating System (OS) the provision of the necessary mechanisms to support these applications. In this paper, an approach for run-time reconfigurable Operating System, which takes advantage of the new hybrid FPGAs to reconfigure itself based on online estimation of application demands, is presented. Especially run-time assignment and reconfiguration of OS services over hybrid architecture are discussed. The proposed model uses a 0-1 Integer programming strategy for assigning OS components over hybrid architecture, as well as an alternative heuristic algorithm for it. Furthermore, the evaluation of the reconfiguration costs are presented and discussed.

1 INTRODUCTION

Nowadays, the usage of Field Programmable Gate Array (FPGA) in the field of Reconfigurable Computing (RC) has become widely used. In particular the capability of a FPGA to be run-time reprogrammed makes its use for reconfigurable systems very attractive Even more attractive is the emerging hybrid

* 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.

FPGAs, which has a hardcore or softcore general purpose processor (GPP) surrounded by a large field of reprogrammable logic. These new components open several interesting possibilities to design reconfigurable architectures for Systems on Chip (SoC). [1].

One of the challenges of our research is to provide support for run-time reconfigurable architectures, which may be used for self-optimizing systems. In dynamic environments, where application requirements may dynamically change, the concept of reconfigurable operating systems appears, which is emerging as new research field.

Differently from the normal approach where the design of such operating system (OS) is done offline, the proposed approach suggests the use of new partial reconfigurable architectures in order to support the development of a hardware/software reconfigurable operating system [2]. In this proposed architecture, the Real-Time Operating System (RTOS) is capable to adapt itself to current application requirements, tailoring the RTOS components for this purpose. Therefore, the system continuously analyze the requirements and reconfigure the RTOS components at the hybrid architecture optimizing the use of system resources. Hence, the system is capable to decide on-the-fly which RTOS components are needed and also to which execution environment (CPU or FPGA) they will be assigned.

The paper focuses on an online partitioning algorithm for a real-time operating system services, which tries to minimize the whole resource utilization and to reduce the reconfiguration costs.

The remaining of the paper is organized as follows: Section 2 presents a brief state-of-the-art analysis regarding hardware implementation of OS services their flexibilities. Then, section 3 shortly presents the architecture used. Section 4 presents the system formulation using 0-1 Integer Programming (BIP) and the reconfiguration costs evaluation. An analysis of the run-time execution of this evaluations, with an heuristic algorithm for the components assignment problem are presented in section 5. Section 6 presents some evaluation results using MATLAB to compare the proposed heuristic algorithm with the original one presented in [3]. Finally, some conclusions and future work are shown in section 7.

2 RELATED WORK

The idea of implementing OS services in hardware is not new. Several works in the literature, [4],[5],[6],[7] and [8]. 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 during run-time even when application requirements may significantly change.

In the field of reconfigurable computing, reconfiguration aspects have been concentrated at application level (see [9], [10] and [11]). At the OS level the research are limited to provide run-time support for those applications (see [6], [12] and [13]).

In the approach presented in this paper we expand the usage of those concepts and the hardware support to the OS level. Additionally, based on the state-of-the-art analysis, for a self-optimized reconfigurable RTOS none such similar approach has been proposed yet.

3 BASIC ARCHITECTURE

Our target architecture is composed of one CPU, configurable logic elements (FPGA), memory and a bus connecting the components. Most of these elements are provided in a single CHIP, such as the Virtex II Pro^{TM}, from Xilinx company. The RTOS services that are able to be reconfigured are stored on an external SDRAM chip in two different implementations: as software object and as FPGA configuration bitstream.

Abstractly, the system can be seen as presented in Figure 1. The target RTOS provide services to the application as a set of hardware and software components. These components may, during run-time, be reallocated (reconfigured) over the hybrid architecture in order to better use the available resources and still meet the application requirements.

Our system concept has a similar approach than a microkernel RTOS, as it is being the concept adopted by most RTOSs. Thus, just absolutely essential core operating system functions are provided by the kernel (which are fixed and can not be reconfigured). The other functionalities (services) are provided by components attached at the kernel. However, these components are reallocated during run-time in order to meet the application requirements and the resource usage constraints.

The usage of microkernel also incorporates the nature advantage of flexibility and extensibility (among others) of it, which is very desired in our case in order to better perform the reconfigurability aspects. Nevertheless, it has the disadvantage to slow down the performance due to the increased necessity of messages changes by the components. Therefore, the Communication Layer presented in Figure 1 performs a efficient communication infrastructure in order to reduce this effect and to offer a transparent set of services to the application (independent of the allocation of the components). The details of the architecture is in its development phase will not be treated in the scope of this paper.

4 PROBLEM DEFINITION

The problem of assigning RTOS components over the two execution environments can be seen as a typical assignment problem. Therefore, we decided to

Fig. 1. Proposed microkernel based architecture.

model the problem using Binary Integer Programming (BIP) [14]. 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 at CPU it represents the processor used by it. Note that these costs are not static, since the application demands are considered to be dynamic. This topic will be addressed later on in subsection 4.2.

4.1 OS Service Assignment

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. As some of the components may not necessary be needed by the current application, they should neither be assigned to the CPU ($j = 1$) nor to the FPGA ($j = 2$). Therefore, to proper represent this situation we consider that this component should stay at memory pool ($j = 3$). As we are focusing on the resource utilization optimization between CPU and FPGA we define that a component i placed on the memory pool ($j = 3$) does not consume any resource ($c_{i,3} = 0$). The definition of a third assignment place for an OS component is more useful for reconfiguration costs estimation, that will be seen in section 4.2.

The resources are limited, which derive two constraints for our BIP formulation: the maximum FPGA area available (A_{max}) and the maximum CPU workload (U_{max}) reserved for the operating system. Thus, the total FPGA area (A) and total CPU workload (U) used by the hardware components and the software components, respectively can not exceed their maximums. These constraints are represented by

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

We also consider that a component i can be assigned just to one of the execution environment: $\sum_{j=1}^{3} x_{i,j} = 1$ for every $i = 1, ..., n$.

To avoid that one of the execution environment would have its usage near to the maximum, we specify a constraint to keep a balanced resource utilization (B) between the two execution environments: $B = |w_1 U - w_2 A| \leq \delta$. Where δ is the maximum allowed unbalanced resource utilization between CPU and FPGA. The weights w_1 and w_2 are used to proper compare the resource utilization between two different execution environments. If the resource used from an execution environment are not near to its maximum, it will have the capability to absorb some variation of the application demands. This characteristic are useful for real-time system in order to avoid the application to miss its deadlines due to workload transients. Note that this approach cannot guarantee hard real time constraints. However, for soft real-time systems it can be considered valid.

The objective function used to minimize the whole resource utilization is defined as

$$min\{\sum_{j=1}^{3}\sum_{i=1}^{n} c_{i,j} x_{i,j}\}$$

The solution of this BIP are the assignment variables $x_{i,j}$, which we define as being a specific system configuration: $\Gamma = \{x_{i,j}\}$.

4.2 Reconfiguration Costs

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 a unbalanced situation ($|w_1 U - w_2 A| > \delta$), the RTOS components should be reallocated in order to bring the system again in the desired configuration. In this situation, not just the new assignment problem need to be solved (Γ') again, but also the costs (time) necessary to reconfigure the system from Γ to Γ' need to be evaluated. This evaluation is necessary since we are dealing with real-time systems. Thus, we have a limited time available for reconfiguration activities.

The reconfiguration cost of every component represents the time necessary to migrate a component from one execution environment to the other one. Therefore, we need to specify for every possible migration of a component its correspondent cost. As it was shown in section 4.1, our model assumes three different environments ($j = \{1, 2, 3\}$). The definition of the environment $j = 3$ (memory pool) is necessary to proper represent the case where a new OS service arrives in the system. This happens when the application requires a service that is neither at CPU nor at FPGA available, but it is stored at the memory pool. The same is valid for a service that leaves the system (it is not more needed by the application). So, we define for a component i a 3×3 size migration costs matrix R_i.

Let $R_i = \{r^i_{j,j'}\}$, where j and j' are the current and new execution environment of component i.

If $x_i = \{x_{i,1}, x_{i,2}, x_{i,3}\}$ and $x'_i = \{x'_{i,1}, x'_{i,2}, x'_{i,3}\}$ are the current and new assignment of the component i, then the complete reconfiguration cost K (total reconfiguration time) of the system is defined as:

$$K = \sum_{i=1}^{n} x_i^T R_i x'_i$$

In our current approach the migration costs associated which a component includes all necessary steps to remove a component from one execution environment to the other one. These steps represents the time to program the FPGA with a component or link the software component with in the CPU programm, translate the context between different execution environments (when necessary), and also read the component instance from memory pool.

5 RUN-TIME ANALYSIS

As our operating system is being designed to support real-time applications, a deterministic behavior for service assignment and system reconfiguration need to be used in order to handle application time constraints.

5.1 Heuristic Algorithm for Assignment Problem

The solution of an BIP finds an optimal solution for the assignment problem. For a small set of components this approach is very suitable. However, it is too computationally complex to solve all problem sizes. Therefore, we are currently using an heuristic greedy based algorithm for this problem. The algorithm is composed by two parts. The first one creates two clusters (FPGA and CPU component sets) from the component set currently needed by the application. The second part improves the first solution towards the balance value (B) and the number of the components to be reconfigured (trying to reduce it). The next paragraphs will concentrate in second part of the algorithm and the details about the first phase can be seen in [3].

The solution given by the first part of the algorithm do not take into consideration the balancing constraint δ. So, there is no guarantee that it will provide a solution which fulfills this constraint. Moreover, it does not take into consideration the reconfiguration costs reduction. Therefore, a second algorithm is proposed that improves the balancing B in order to meet the δ constraint. It is based on Kernighan-Lin algorithms [15] and it aims to obtain a better balancing B than the first one by swapping pairs of components between CPU and FPGA. It also tries, to minimize the number of components being reconfigured in order to reduce the total reconfiguration cost. The algorithm receives as input the first assignment solution X which has $nc_1 = \sum_{i=1}^{n} x_{i,1}$ components assigned to CPU and $nc_2 = \sum_{i=1}^{n} x_{i,2}$ components assigned to FPGA. The maximum number of pairs that are possible to be swapped is defined as: $max\_pairs = min(nc_1, nc_2)$.

By moving a component i, previously assigned to the CPU, to the FPGA ($\{x_{i,1} = 1; x_{i,2} = 0\} \Rightarrow \{x_{i,1} = 0; x_{i,2} = 1\}$), we have a new balancing B: $B_{new} = |B_{current} - s_i|$, where $s_i = \{c_{i,1} + c_{i,2}\}$. Similarly, by moving a component i from FPGA to CPU, the new balancing B will be: $B_{new} = |B_{current} + s_i|$. Thus, swapping a pair of components o, p ($\{x_{o,1} = 1; x_{o,2} = 0\}; \{x_{p,1} = 0; x_{p,2} = 1\}$), the new balancing B is defined as: $B_{new} = |B_{current} - s_o + s_p|$. Similarly, $B_{new} = |B_{current} + s_o - s_p|$ if $\{x_{o,1} = 0; x_{o,2} = 1\}; \{x_{p,1} = 1; x_{p,2} = 0\}$. Additionally, we define G_{op} as the gain obtained in the balancing B by swapping a pair o and p of components: $G_{op} = B_{new} - B_{current}$. A gain below 0 means an improvement obtained in the balancing B.

The reconfiguration costs reduction is executed indirectly by means of reducing the number of components to be reconfigured. Therefore, a function $\Delta X = diff(X^a, X^b)$ (where $\Delta X = \{\delta x_i\}$, $X^a = \{x^a_{i,j}\}$ and $X^b = \{x^b_{i,j}\}$) is used to give the information if a component x_i has different allocation in X^a and X^b. Thus, the function $diff$ is defined as follows:

$$\delta x_i = \begin{cases} 1 & : \texttt{if } \{x^a_{i,1}; x^a_{i,2}\} \neq \{x^b_{i,1}; x^b_{i,2}\} \\ 0 & : \texttt{otherwise} \end{cases}$$

Algorithm 1 Balancing B improvement and reconfiguration cost reduction

$X^{init}_1 = \{x_{i,1}\}$ Initial assignment of CPU components
$X^{init}_2 = \{x_{i,2}\}$ Initial assignment of FPGA components
$X^{init} = X^{init}_1 \cup X^{init}_2$; $X^{new} = X^{init}$;
$B^{init} = |U^{init} - A^{init}|$; $B^{new} = B^{init}$;
$X^{orig} =$ *Current System Configuration* Γ
$m = max\_pairs$ maximum number of pairs
for $k = 1$ to m **do**
  Find the pair o, p ($\{x_{o,1} = 1; x_{o,2} = 0\}$;$\{x_{p,1} = 0; x_{p,2} = 1\}$ or $\{x_{o,0} = 0; x_{o,2} = 1\}$;$\{x_{p,1} = 1; x_{p,2} = 0\}$) so that o and p are unlocked and G_{op} is minimal
  if $G_{op} < 0$ **then**
    Swap o and p and test it $\Rightarrow X^{try} = (X^{new}$ with o and p swapped)
    $\Delta X = diff(X^{orig}, X^{try})$
    if $\delta x_o = 0$ OR $\delta x_p = 0$ **then**
      Update the new configuration $\Rightarrow X^{new} = X^{try}$
      $B^{new} = B^{new} + G_{op}$
      if $\Delta X_o = 0$ **then** Lock o **end if**
      if $\Delta X_p = 0$ **then** Lock p **end if**
    else
      if $x^T_o R_o x'_o < x^T_p R_p x'_p$ **then** Lock o **else** Lock p **end if**
    end if
  end if
  if $G_{op} \geq 0$ OR all pairs are locked **then break**
end for
return X^{new}

The algorithm starts trying to swap all possible pairs and storing the gain obtained by every try. It than chooses the one which provides the smallest gain. If this gain is bigger than or equal to zero, none swap is able to provide an improvement in the balancing B and the algorithm stops. Otherwise, the pair is swapped and locked according to some rules. If, at least, one of the components from the pair keeps its position in relation to the current system configuration, the pair swap is allowed. In addition, the component that preservers its position (or both) are locked (no longer a candidate to be swaped). However, if both components of the pair change their positions in relation to the current system configuration, no swap occurs. Moreover, just one component (which provides the smaller reconfiguration cost) is locked. This lock is necessary, otherwise the algorithm would not terminate. This process is then repeated until all pairs have been locked or no improvement can be obtained by any interchange. By applying those rules, the algorithms tries to reduce the numbers of components needed to be reconfigured. Also note that the algorithm does not terminate if the δ constraint is fulfilled. This enforce the search for more components (pairs) that could be kept in its current allocation solution.

The algorithm terminates by returning the new assignment solution X that provides a better (or at least an equal) balancing B than the solution provided by the first one. In addition, the number of components being reconfigured is reduced. The complexity of the balancing improvement algorithm is (worst case) $O(m^3)$, where m is the maximum number of pairs. This is due to the fact that we have one *for* loop (1 to m) where in each loop interaction the combination of all components assigned to the CPU with all components assigned to the FPGA are tested (m^2). The algorithm for balancing improvement is shown in Algorithm 1.

6 EXPERIMENTAL RESULTS

For system evaluation of the run-time assignment problem, we made some simulations using MATLAB tool. The results achieved by the original balance improvement algorithm (published in [3]) and the improved one (presented in this paper) were compared. 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 absolute difference cost ($|w_1U - w_2A|$) and the number of components being reconfigured achieved each version of the balancing improvement algorithm 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 current system configuration considered was the previous random system generated.

The Figure 2 shows the results of the balance improvement achieved by the original algorithm (Heuristic-2a) and the optimized one (Heuristic-2b). The improvement made in both case were satisfactory. Note that the results achieved

Fig. 2. Unbalance average for different δ constraints.

by Heuristic-2b, concerning the balance, are quite under the constraint δ. This is due to the fact that the Heuristic-2a algorithm still search for more pairs to be swapped, even with the δ constraint being fulfilled, in order to reduce the number of reconfigurations. This effect can be seen in Figure 3.

Fig. 3. Number of components being reconfigured for different δ constraints.

7 CONCLUSIONS AND FUTURE WORK

In this paper we have presented our investigation towards a run-time reconfigurable RTOS running over a hybrid platform, focusing in the OS service assignment and system reconfiguration. Looking at the related work, we are quite convinced that this is a novel approach for a self-optimized RTOS.

A shortly presentation of the concept of our architecture was also presented. The 0-1 Integer Programming model of the system and the reconfiguration cost

evaluation have been presented. Additionally, considerations of a run-time execution of this technics, in order to support real-time applications have been discussed.

As a future work, the investigation of a proper OS components assignment algorithm which takes into consideration the application time constraints and the integration of the communication costs among the components are going to be made. Moreover, the schedule of the components reconfiguration using technics of RTOS Scheduling, necessary to guarantee the application time requirements are going to be integrated.

References

1. Andrews, D., Niehaus, D., Ashenden, P.: Programming models for hybrid cpu/fpga chips. Computer - Inovative Thecnology for Computer Professionals (2004) 118–120 IEEE Computer Society.
2. Götz, M.: Dynamic hardware-software codesign of a reconfigurable real-time operating system. In: Proc. of ReConFig04, Mexican Society of Computer Science, SMCC (2004)
3. Götz, M., Rettberg, A., Pereira, C.E.: Towards run-time partitioning of a real time operating system for reconfigurable systems on chip. In: Proceedings of International Embedded Systems Symposium 2005, Manaus, Brazil (2005)
4. Lee, J., III, V.J.M., Ingstrm, K., Daleby, A., Klevin, T., Lindh, L.: A comparison of the rtu hardware rtos with a hardware/software rtos. In: ASP-DAC2003, (Asia and South Pacific Design Automation Conference). (2003) 6 Japan.
5. Kuacharoen, P., Shalan, M., Mooney, V.: A configurable hardware scheduler for real-time systems. In: Proc. of ERSA. (2003)
6. Kohout, P., Ganesh, B., Jacob, B.: Hardware support for real-time operating systems. In: International Symposium on Systems Synthesis, Proceedings of the 1st IEEE/ACM/IFIP International conference on HW/SW codesign and system synthesis (2003)
7. Lee, J., Ryu, K., III, V.J.M.: A framework for automatic generation of configuration files for a custom hardware/software rtos. In: Proc. of ERSA. (2002)
8. Lindh, L., Stanischewski, F.: Fastchart - a fast time deterministic cpu and hardware based real-time-kernel. In: EUROMICRO'91. (1991) 12–19 Paris, France.
9. Harkin, J., McGinnity, T.M., Maguire, L.P.: Modeling and optimizing run-time reconfiguration using evolutionary computation. Trans. on Embe.Comp.Sys. (2004)
10. Quinn, H., King, L.A.S., Leeser, M., Meleis, W.: Runtime assignment of reconfigurable hardware components for image processing pipelines. In: FCCM. (2003)
11. Mignolet, J.Y., Nollet, V., Coene, P., Verkest, D., Vernalde, S., Lauwereins, R.: Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In: DATE. (2003)
12. Wigley, G., Kearney, D.: The development of an operating system for reconfigurable computing. In: FCCM. (2001)
13. Walder, H., Platzner, M.: A runtime environment for reconfigurable hardware operating systems. In: FPL. (2004)
14. Wolsey, L.A.: Integer Programming. Wiley-Interscience (1998)
15. Eles, P., Kuchcinski, K., Peng, Z.: 4. In: System Synthesis with VHDL: A Transformational Approach. Kluwer Academic Publishers (1998)