Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 121353, 16 pages
doi:10.1155/2011/121353

Research Article

Operating System for Runtime Reconfigurable Multiprocessor Systems

Diana Göhringer,[1] Michael Hübner,[2] Etienne Nguepi Zeutebouo,[1] and Jürgen Becker[2]

[1] *Object Recognition Department, Fraunhofer IOSB, 76275 Ettlingen, Germany*
[2] *ITIV, Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany*

Correspondence should be addressed to Diana Göhringer, diana.goehringer@iosb.fraunhofer.de

Received 20 August 2010; Revised 30 January 2011; Accepted 14 February 2011

Academic Editor: Aravind Dasu

Operating systems traditionally handle the task scheduling of one or more application instances on processor-like hardware architectures. RAMPSoC, a novel runtime adaptive multiprocessor System-on-Chip, exploits the dynamic reconfiguration on FPGAs to generate, start and terminate hardware and software tasks. The hardware tasks have to be transferred to the reconfigurable hardware via a configuration access port. The software tasks can be loaded into the local memory of the respective IP core either via the configuration access port or via the on-chip communication infrastructure (e.g. a Network-on-Chip). Recent-series of Xilinx FPGAs, such as Virtex-5, provide two Internal Configuration Access Ports, which cannot be accessed simultaneously. To prevent conflicts, the access to these ports as well as the hardware resource management needs to be controlled, e.g. by a special-purpose operating system running on an embedded processor. For that purpose and to handle the relations between temporally and spatially scheduled operations, the novel approach of an operating system is of high importance. This special purpose operating system, called CAP-OS (Configuration Access Port-Operating System), which will be presented in this paper, supports the clients using the configuration port with the services of priority-based access scheduling, hardware task mapping and resource management.

1. Introduction

Scheduling of tasks within a given time frame and with respect to a required deadline due to real-time aspects is well known in computer science from operating systems (OSes), especially in real-time operating systems (RTOSes). Scheduling strategies of conventional OSes vary between pre-emptive and non-pre-emptive scheduling. They can be further classified into static and dynamic scheduling, where static scheduling occurs at design time and dynamic scheduling at runtime. Therefore, dynamic scheduling is more suitable for runtime adaptive systems. Well-known dynamic scheduling algorithms are earliest deadline first (EDF) or rate monotonic algorithm (RMA) (see [1] for detailed descriptions). The classical scheduling and task mapping process of software-based systems with a traditional OS has its counterpart in novel runtime reconfigurable hardware systems. Within these systems, tasks can be presented additionally to the traditional software representation, as physical hardware realization, for example, on an FPGA. That means that an additional degree of freedom for task mapping on hardware resources is available for the OS layer. For example, compared to a task in a traditional software-based system that was mapped and executed on a resource as a software thread, the hardware reconfigurable variant of such a system would also allow running this task as a hardware block realized with logic resources on an FPGA. This difference and the new degree of freedom in task representation require the consideration of a novel concept for hardware task scheduling and mapping. In order to handle this process, a detailed analysis of the consequences, for example, due to data dependencies, priority, and real-time aspects, has to be investigated and formalized into a feasible algorithm for an efficient, special-purpose OS. Furthermore, the underlying hardware resources, including the internal configuration access port (ICAP), have to be

characterized in terms of timing, determinism, behavior in termination cases, and so forth. Also, these results have to be accounted for in the special-purpose OS approach by a cost function. The described investigation and the results can be exploited efficiently in the runtime adaptive multiprocessor system-on-chip (RAMPSoC) approach as described in [2]. In this approach, several processors, coprocessors, and hardware accelerators are available for concurrent task realization on an FPGA. The approach presented in this paper allows scheduling tasks of a control dataflow graph (CDG) and mapping these tasks either in hardware or in software on a reconfigurable multicore system on the FPGA. The algorithm, therefore, considers data dependencies; physical constraints from the configuration interface and the reconfigurable resources; the capability of the parallel data processing hardware of the RAMPSoC approach.

The paper is organized as follows: related work is presented in Section 2. Section 3 describes briefly the RAMPSoC approach and its features. In Section 4, the concept and the features of CAP-OS (configuration access port-operating system) are described. Section 5 presents how CAP-OS is integrated into the RAMPSoC hardware architecture. The implemented system and first results are presented in Section 6. A case study with an image-processing application is shown in Section 7. Finally, the paper is closed by presenting the conclusions and an outlook in Section 8.

2. Related Work

Scheduling for a hardware reconfigurable architecture is used in approaches reported in various publications. The selected publications discussed in this paper are only a subset of the numerous approaches developed in academic and industrial environment. However, the selected papers reflect the significant aspects in respect to the presented approach and allow an objective comparison of the benefits achieved in the proposed solution of the special-purpose OS named CAP-OS.

Garcia et al. [3] give an overview of the requirements for runtime- and operating systems for reconfigurable hardware-based systems. The authors especially point out the fact that physical constraints, such as the availability of hardware resources (in terms of area) and the configuration time, limited by the bandwidth of the configuration memory and interface, have to be taken into account for the scheduling. Especially, these are the challenges which have to be taken into account when reconfigurable hardware aware operating systems are introduced or developed.

Dittmann and Frank [4] describe a scheduling approach for a single processor and several accelerators, which can be configured at runtime. The solution provides a pre-emptive reconfiguration, which is important if a task with a higher priority has to substitute the configuration process of a task with lower priority. The scheduling strategy is based on a deadline monotonic (DM) algorithm with some extensions related to the fact that a hardware/software reconfigurable system is targeted. The approach has some restrictions due to the fact that only homogeneously shaped reconfigurable areas are supported. Because of this, only a fixed time frame for reconfiguration of the hardware is considered in the algorithms. In real systems, especially, when different hardware IPs have to be reconfigured, this time can vary significantly. A further restriction is that data dependencies between the tasks are not considered within the scheduling algorithm. The CAP-OS approach incorporates this into the metrics for the scheduling in order to achieve a beneficial scheduling of the hardware tasks. Furthermore, the approach requires drivers supporting the physical reconfiguration of the FPGA. This certainly could be a standard ICAP driver with the related IP cores.

Ullmann et al. [5] also target a single-processor solution with reconfigurable accelerators in a homogeneous shape and size, similar to the previously described approach. The scheduling is priority based and non-pre-emptive due to the fact that this approach was developed for automotive applications where pre-emption of a certain task is not allowed. The reported runtime system in the paper includes the hardware drivers for the configuration access port. The runtime system included some features, such as context load and save, which allows the resumption of tasks in hardware or software, or even a migration of the tasks from hardware to software or vice versa. The restrictions of this approach are mainly in the high overhead if a different application scenario needs to be realized. A time-consuming and hand crafted adaptation of the runtime system needs to be done. Furthermore, the fact that this approach was developed for the automotive domain limits the reuse in other application domains, such as image processing, where a more flexible scheduling is required.

ReconOS [6] uses an eCos (embedded configurable operating system) real-time operating system as basis for the scheduling approach. Also, here a single processor with loosely coupled reconfigurable accelerators is the target hardware architecture. In comparison to the previously described approach, the authors use a fixed priority scheduling approach. For synchronization purposes, a communication method for the software and hardware threads over the eCOS RTOS was developed. An interesting result is that a task graph with dependent and independent tasks is used as the input description for the scheduler. However, the limitation of using a single processor to perform the applications differs the ReconOS approach from CAP-OS. In CAP-OS, a variety of processors and accelerators can be handled.

In [7], the authors describe a concept for a quality of service- (QoS-) based operating system for multimedia applications on hand-held computers (e.g., PDA) with a reconfigurable accelerator. The paper includes also the strategy to abstract from the hardware layer in order to hide the complexity of the heterogeneous architecture consisting of a network-on-chip (NoC) and the processing elements from the developer. In relation to the RAMPSoC approach, the solution does not include a heterogeneous and adaptive communication infrastructure and processors and does not include the design flow for generating the required sources. The paper describes a definitely pioneering work in the area of operating systems for reconfigurable

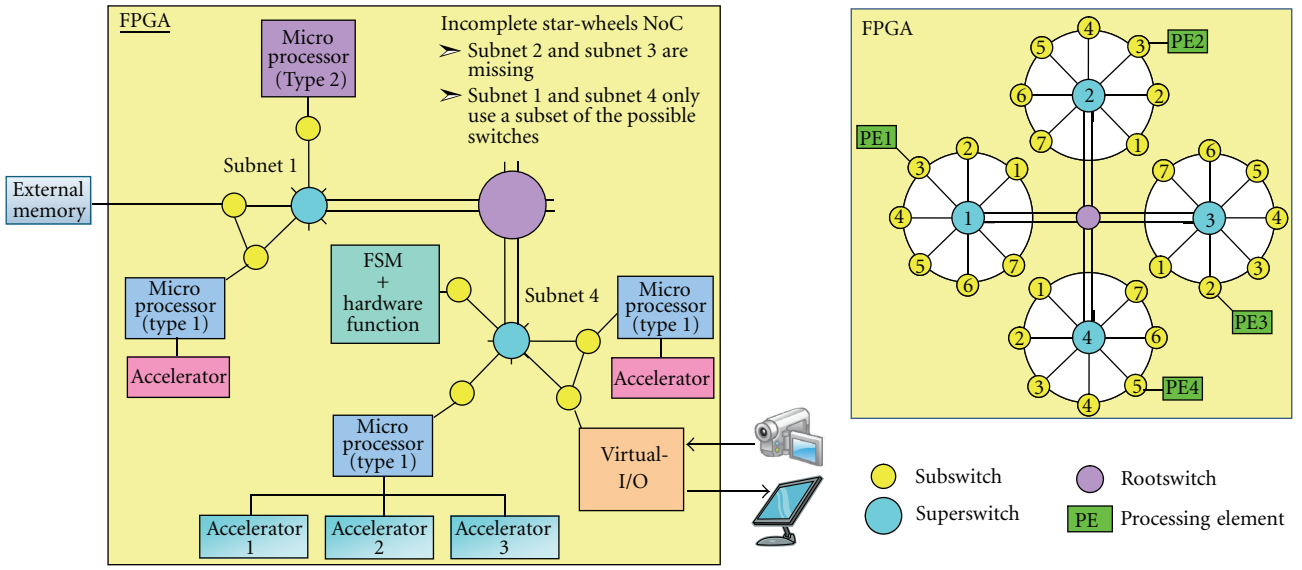


Figure 1: Example of an RAMPSoC architecture connected over an incomplete star-wheels network-on-chip. For comparison, on the top right side, an example of a complete star-wheels NoC is illustrated.

hardware and can be seen as the root for the evolution in this domain.

Based on the reported approaches, it is obvious that a novel OS approach for a runtime reconfigurable multiprocessor systems, such as RAMPSoC, has to be developed and introduced. One simple example for this necessity is the fact that the reconfigurable regions are no longer homogeneous in their footprint and, therefore, the configuration times vary between the different tasks, which may have to be allocated to the hardware. This and other parameters have to be handled with the novel approach of the CAP-OS.

3. The RAMPSoC Approach

The CAP-OS is used for runtime scheduling, task mapping, and resource management on a runtime reconfigurable multiprocessor system, such as RAMPSoC [2]. Figure 1 shows an example for a RAMPSoC architecture at one point in time. RAMPSoC is a heterogeneous multiprocessor system-on-chip (MPSoC) with distributed memory. It consists of a number of different processors connected over a communication infrastructure, which is a heterogeneous network-on-chip (NoC) called star-wheels NoC [8] in this example. Between others, the advantages of the star-wheels NoC are that it supports runtime adaptation and that it does not need to be implemented completely. Therefore, if additional switches are demanded, they can be added at runtime. Furthermore, different clock domains are supported, which is important, to achieve a good performance per watt ratio for multiprocessor systems. Additionally, a high throughput and therefore a low latency are supported, which is important for, for example, image-processing applications. Depending on the application requirements and the number of needed processors, also other communication infrastructures, such as point-to-point connections, buses, or other NoCs are supported and can be selected from a library at design time.

Each processor can be extended with one or several hardware accelerators to increase their performance for special-purpose instructions. Also, a finite-state machine (FSM) together with a hardware function can be used instead of a processor. The FSM is required to support the communication protocol over the NoC for communicating with the other processing elements.

Different processors can be chosen from a library (e.g., Xilinx MicroBlaze [9], Leon Sparc [10], etc.), and also a small library for image-processing hardware accelerators exists.

Dynamic and partial reconfiguration [11] is used to adapt the RAMPSoC hardware architecture at runtime. The software executables for the processors can be loaded at runtime either also by exploiting dynamic and partial reconfiguration (similar to the approach described by Sander et al. [12]) or by transferring the software executables via the communication infrastructure (e.g., the NoC). Furthermore, the clock frequency of the different processing elements can also be adapted at runtime. Like for the software executables, also here two possibilities exist: either by reconfiguring the appropriate digital clock manager (DCM) [13] or by switching to a different clock domain. In summary, the following runtime adaptations are supported by RAMPSoC:

(i) number and characteristics of processors,

(ii) communication infrastructure (e.g., size, bandwidth, and topology),

(iii) number and functionality of hardware accelerators,

(iv) software executables of the processors,

(v) clock frequency of the processing elements and network domains.

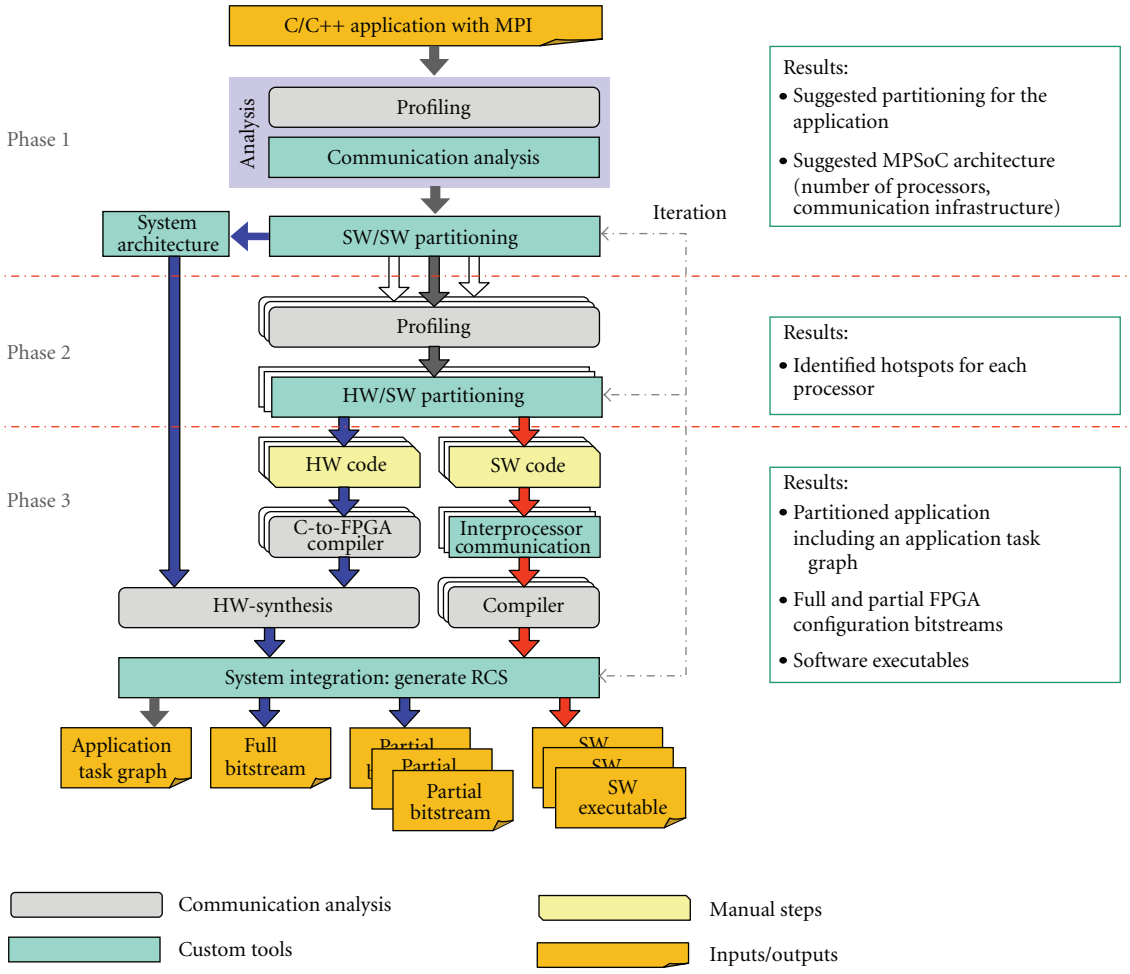


Figure 2: Design methodology of RAMPSoC.

A well balance between performance, power consumption, and area requirements can be achieved through runtime adaptation of the hardware architecture in respect to the requirements of the applications. More details about the hardware architecture of the RAMPSoC and its benefits can be found in [2].

For an efficient programming of such a flexible hardware architecture, an easy-to-use design methodology is required, which guides the user in application partitioning and in generating the appropriate hardware architecture at design time. As a result of the different analysis and partitioning steps, a task graph is generated for each partitioned application. The design suite also generates the partial bitstreams for the several hardware modules (e.g., processors, accelerators) as well as the software executables for the different processors. Figure 2 shows an overview of the current status of the design flow, which can be used for normal C/C++ applications or C/C++ applications using the message passing interface (MPI) [14]. MPI is a standard parallel programming model, which is used for supercomputing applications and especially to program multiprocessor systems with distributed memory. As the processors have only local memory, they exchange information by sending messages using the MPI standard protocol. RAMPSoC has its own MPI implementation layer, which translates the MPI standard protocol commands into the appropriate communication protocol required by the star-wheels NoC. The support for further possible communication infrastructures is currently under development as well as the improvement of the design methodology. A more detailed description of the functionality of the different tools within the design methodology can be found in [15].

The partial bitstreams, the software executables, and the task graphs of the applications are required by the CAP-OS, which will be presented in detail in the next section. The CAP-OS is responsible for the runtime scheduling of the configurations of the different tasks, allocating the tasks to the processing elements and for resource management. Furthermore, the CAP-OS needs to respond to runtime demands of the application, such as one or several processors requesting additional or different accelerators.

4. Concept of the CAP-OS

For an adaptive MPSoC, such as RAMPSoC, a flexible RTOS is required, which schedules the reconfiguration of the tasks and their runtime allocation to a specific processing

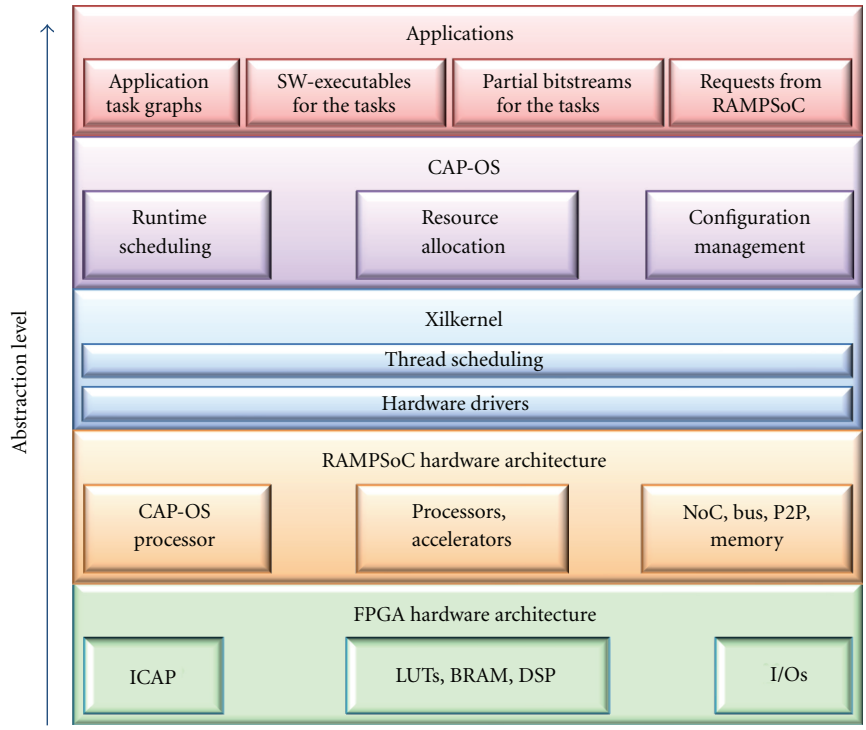


Figure 3: CAP-OS embedded in the several abstraction layers of the system approach.

element. Furthermore, this RTOS has to assure that the different applications meet their real-time requirements and that the utilization of the hardware resources and therefore the power consumption is kept low. Figure 3 shows how the CAP-OS manages the underlying RAMPSoC hardware architecture to fulfill the real-time requirements of the user applications. The CAP-OS further hides the complexity of the underlying dynamic RAMPSoC architecture from the user.

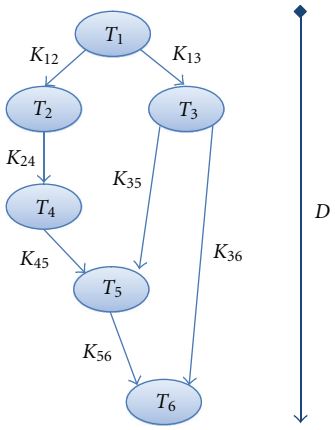
Physical resource allocation at runtime is done by performing partial and dynamic reconfiguration using the ICAP. Software task can be loaded either using the ICAP or using the interprocessor communication architecture. Therefore, the scheduling algorithm has to consider the time required for reconfiguring/loading a module, which depends on the data throughput of the ICAP interface or the communication infrastructure and certainly on the size of the module. This time frame is not negligible since the data amount for hardware modules can be very small, but also several hundred kilobytes. The software modules are normally smaller than 250 Kbytes, due to the restricted on-chip memory, while the hardware modules can be bigger. For each task, two different implementation options exist. A task can either be executed in software on a processor or in hardware as a hardware accelerator. The hardware task is normally a codesign, where parts of the task are executed in software on the processor and the compute intensive part is executed in hardware with the closely coupled accelerator. The choice to implement a task in software or in hardware depends normally on different parameters such as varying allocated hardware area, performance, and reconfiguration/loading time. The scheduling algorithm has to choose the appropriate type of realization to fulfill the real-time constraints. Moreover, the presented scheduling approach tries to reuse existing resources, which were already configured onto the chip in a previous point of time, with the goal to reduce the overall reconfiguration overhead. Furthermore, the scheduling algorithm has to support preemptive reconfiguration, because while reconfiguring one task, it can happen that a request for the reconfiguration of another task with higher priority occurs. As only one ICAP is available, the reconfiguration of the previous task has to be terminated and the new task needs to be reconfigured. After this procedure, the reconfiguration of the interrupted task has to restart, because a continuation of the terminated reconfiguration is not supported by the FPGA vendor. In contrary to this, the loading of an SW task via the communication infrastructure can be preempted and resumed. Here, a restart as for the ICAP interface is not required. The communication infrastructure allows the loading of more than one software task simultaneously, but then the bandwidth will be divided between the tasks. Therefore, it is more beneficially to load one task after the other based on the priorities of the tasks.

The scheduling approach presented in this paper can handle both independent and dependent tasks. A group of interrelated tasks is called a task graph (TG). Each TG must fulfill the following requirements:

(i) the TG is a directed acyclic graph (DAG),

(ii) each task runs on processors/hardware accelerators,

(iii) each task has an identity (ID)



T_x: task x
D: global deadline
K_{xy}: communication costs between task x and task y

Figure 4: Example task graph with global deadline, interrelation, and communication costs.

(iv) each task has the following information:

(1) neighborhood relation (predecessor/successor),
(2) algorithm type or hardware constraints (Algo-ID),
(3) execution time, reconfiguration/loading time,
(4) communication costs,
(5) name of the corresponding partial bitfile or software executable,

(v) the TG has a global deadline (D),

(vi) the TG has either hard or soft real-time constraints, which are inherited by the tasks belonging to the TG.

For the configuration of a task, the following three rules apply:

(i) it can be terminated;

(ii) it can be interrupted (only for SW tasks loaded over the communication infrastructure);

(iii) it is only feasible, after all predecessor tasks are completely reconfigured/loaded.

Figure 4 illustrates an example of such a TG including the global deadline, the interrelation, and the communication costs.

Within the CAP-OS, each task within a TG has a life cycle as shown in Figure 5.

Table 1 describes each of the states, which are traversed by a task during its life cycle, in detail.

Important here is that if the configuration of a task is interrupted, the task returns into the *Ready* state, the configuration data is lost and has to start all over again. The loading of a software task via the communication infrastructure on the other hand can be resumed if it has been interrupted. The address of the last word, which has been transferred via the communication infrastructure to the target processor, is stored, so that CAP-OS can later resume loading the software executable. Exceptions are

(i) the interrupted software will be remapped to a different processor;

(ii) the software had been interrupted by another software task, which will be loaded on the same processor. After the task has finished executing, the interrupted software will be loaded again on this processor.

In both cases, the loading of the interrupted software task has to restart from the beginning.

For simplicity, in the following section, the terms *configuration* and *configuration time* will be used for both configuration via ICAP and loading via the communication infrastructure. Loading via the communication infrastructure will only be used when explicitly requested.

As already mentioned in the previous section, the multiprocessor model used for the scheduling is a heterogeneous runtime adaptive MPSoC that uses a message passing communication scheme. The runtime scheduling algorithm is only performed for tasks, which are in state *Ready*. The novel runtime scheduling approach is described in detail in the next subsection.

4.1. The Novel Runtime Scheduling Approach. The novel runtime scheduling algorithm is divided into two main steps. First, a static scheduling algorithm is used to roughly assign priorities to the tasks of each TG using the information given by the TG description. The TG description has been received together with the bitstreams and software executables from the RAMPSoC design methodology. The TG description is written using the XML standard format and includes the following information: list of all tasks and detailed information for each task. For each task, these files contain the ID, the algo type, the successor tasks, the communication costs, the name of the bitstream file or/and software executable, the reconfiguration/loading time, and the execution time. Furthermore, for each task graph, the global deadline is given. Finally, this file also includes a list of possible processors, their configurations (e.g., pipeline length, memory size, specialized instructions, etc.), and the name of the corresponding bitfile. CAP-OS parses this XML file and updates its internal tables for the tasks and the processors. Also it stores the bitstreams and software executables, which have been received by the user, for example, via a Compact Flash card or an Ethernet connection, in the external memory.

For the priority-based static scheduling, the list scheduling algorithm is used, because it respects resource constraints. The available resources are the single ICAP, the communication infrastructure, and the maximum number of possible processors, which depends on the size of the chosen FPGA. First conservative estimates for the ASAP (as soon as possible) and the ALAP (as late as possible)

Table 1: Description of the life cycle states of a task.

Configuration and execution	Description
Not_ready	This task is not ready for reconfiguration, because its predecessors are not completely reconfigured.
Ready	This task is ready for reconfiguration and competes with the other *Ready* task for the access to the ICAP. Only tasks without predecessors, or whose predecessors have already been reconfigured, can enter this state.
Config	The task is under configuration/loading via the ICAP/the communication infrastructure onto the RAMPSoC. If a task with higher priority becomes *Ready*, the reconfiguration/loading process is terminated/interrupted, and the task returns into the *Ready* state and waits for a new possibility to access the ICAP/communication infrastructure.
Exec	After successful configuration/loading, the task starts execution and enters this state. An execution cannot be interrupted.
Exit	After the execution, the task enters this state. The allocated processing element is now free for the next task.

Figure 5: Life cycle states of a task.

start times for each task of a TG, consisting of m tasks, are calculated using the following formulas:

$$\begin{aligned} \mathrm{ASAP}(T_x) &= \sum_{T \in \mathrm{pre}(T_x)} (t_{\mathrm{con}}(T) + t_{\mathrm{exe}}(T)), \\ \mathrm{pre}(T_x)&\text{: Predecessor of task } T_x, \\ t_{\mathrm{con}}(T)&\text{: Configuration time of task ,} \\ t_{\mathrm{exe}}(T)&\text{: Execution time of task } T, \end{aligned} \tag{1}$$

$$\begin{aligned} \mathrm{ALAP}(T_x) &= D - \sum_{T \in \mathrm{succ}(T_x)} (t_{\mathrm{con}}(T) + t_{\mathrm{exe}}(T)), \\ \mathrm{succ}(T_x)&\text{: Successor of task } T_x, \\ D&\text{: Global deadline of the task graph,} \end{aligned} \tag{2}$$

$$\begin{aligned} \mu(T_x) &= \mathrm{ALAP}(T_x) - \mathrm{ASAP}(T_x), \\ \mu(T_x)&\text{: Mobility of task } T_x. \end{aligned} \tag{3}$$

The task loading via the communication infrastructure is the default procedure for software task to keep the ICAP interface available for the hardware task, because they do not have an alternative data path. Only when the communication infrastructure is blocked with a high-priority task, and another high-priority software task needs to be loaded as well, then the ICAP interface will be used to load the second software task. At this moment, it is not known, if an already configured processor can be reused for the software task. Therefore, t_{con} is the time required to reconfigure a new processor together with the software required for this task. This results in the worst case ALAP and ASAP start time.

Based on the ASAP and ALAP start time of each task, a priority can be assigned to each task in the TG using the urgency or the mobility of each task. The urgency depends on the maximum number of successors of a task. The mobility of a task (see Formula (3)) is the difference between its ALAP and ASAP start time and favors the tasks along the critical path. The TG in Figure 6 has, for example, the following critical path: $T1 \rightarrow T2 \rightarrow T4 \rightarrow T5 \rightarrow T6$. Because of this, the mobility is used here to assign the priorities to the tasks. The smaller the mobility, the higher is the priority of the task.

At runtime, only the *Ready* tasks are scheduled for configuration according to their priorities, which have been calculated with the list scheduling algorithm. Figure 6 shows such a TG which is processed by the CAP-OS for the purpose of scheduling the configuration of the different tasks. In the current time step shown in Figure 6, $T1$ has already been configured, and therefore $T2$ and $T3$ are now in the *Ready* state. Normally, the task with the highest priority will be configured first. If there are two or more *Ready* tasks and the difference between the mobility of the two tasks with the highest priority is smaller than the configuration time of the task with the lower priority (see Formula (4)), a dynamic cost function $K(T_x, T_y)$ (Formula (5)) is used to reassign the priorities of these two tasks.

$K(T_x, T_y)$ considers the ratio between the mobility of the two tasks $K_1(T_x, T_y)$ (Formula (6)) and the ratio between the number of successors of the two tasks $K_2(T_x, T_y)$ (Formula (7)). $K(T_x, T_y)$ is computed using Formulas (5) to (7), and it is only computed for the current two tasks with the highest priority to be scheduled. K_1 gets a greater weight in the cost function compared to K_2, because for real-time applications the execution time is the most important factor. Therefore, the default values were set to 0.6 for ω_1 and 0.4 for ω_2. These weights can be modified by the user depending on the requirements of the application. Additionally, multiple TGs can be scheduled at runtime. If some of these TGs have hard real-time and others only soft real-time requirements, then all tasks of the TGs with the soft real-time constraints will be delayed. They will be configured after the tasks with the hard real-time constraints, even though they might have a higher priority according to the list scheduling algorithm. This is

important, to assure that the hard real-time TGs meet their constraints.

T_x gets highest priority if

$$\mu(T_y) - \mu(T_x) > CT(T_y), \quad \mu(T_x) < \mu(T_y),$$
$$\mathrm{CT}(\mathrm{T_y}): \text{Configuration time of task } y. \tag{4}$$

Else decision is made using $K(T_x, T_y)$:

$$\begin{cases} K(T_x, T_y) < 0, & T_y \text{ gets highest priority,} \\ K(T_x, T_y) \geq 0, & T_x \text{ gets highest priority,} \end{cases}$$
$$K(T_x, T_y) = \omega_1 * \left(K_1(T_x, T_y) - K_1(T_y, T_x)\right) + \omega_2 * \left(K_2(T_x, T_y) - K_2(T_y, T_x)\right), \tag{5}$$
$$\omega_1, \omega_2: \text{ Weighting factors,}$$

$$K_1(T_x, T_y) = \begin{cases} \dfrac{\mu(T_y)}{\mu(T_x)}, & \mu(T_x) < \mu(T_y) \wedge \mu(T_x) \neq 0, \\ 0, & \text{else} \end{cases}$$
$$\mu(T_x): \text{ Mobility of task } x, \tag{6}$$

$$K_2(T_x, T_y) = \begin{cases} \dfrac{N(T_x)}{N(T_y)}, & N(T_x) > N(T_y) \wedge N(T_y) \neq 0, \\ 0, & \text{else} \end{cases}$$
$$N(T_x): \text{ Number of successors of task } x. \tag{7}$$

Finally, an additional feature is supported by CAP-OS. This feature allows for increasing the clock frequency of a processing element at runtime. This can be done by either reconfiguring the corresponding digital clock manager (DCM) [16] or by using clock multiplexers to switch to a different clock frequency. Both approaches have been described in [17]. The reconfiguration of a DCM takes more time than the use of clock multiplexers. However, the DCM reconfiguration provides a greater variety of possible clock frequencies than the clock multiplexers, because only a limited number of clock multiplexers are available on the FPGA. Both approaches are supported here. Therefore, in the following, DCM reconfiguration stands for both approaches. The user has to select at design time the approach, which is more appropriate for the target application.

DCM reconfiguration is used to speed up the execution time of a task. Hereby, it is assumed that the execution time stays in strong relation to the clock frequency. This DCM reconfiguration is used if a task cannot complete within its ALAP time or if another task urgently requires the same processor.

Therefore, the single steps of the CAP-OS scheduling algorithm can be summarized as follows:

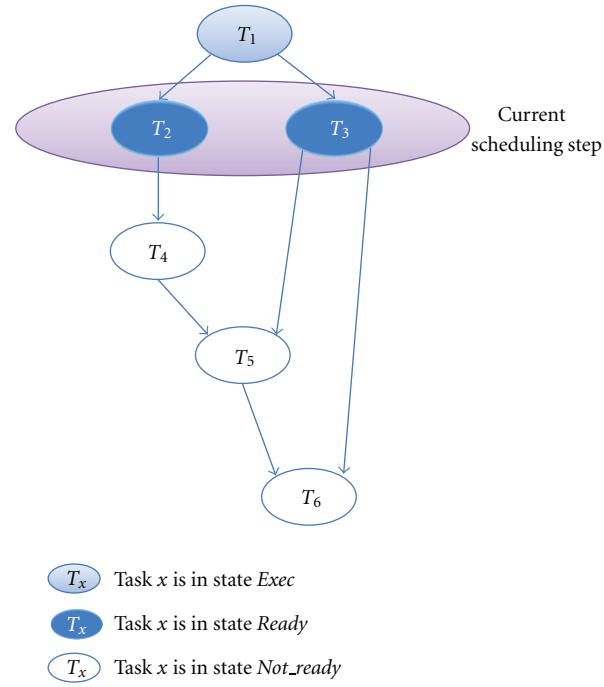


Figure 6: Task graph to illustrate the functionality of the scheduling.

(1) calculate ASAP and ALAP start time for each task in the task graph,

(2) calculate the mobility of each task and schedule their priorities using a list scheduling algorithm,

(3) select the *Ready* tasks, and schedule them dynamically:

   (a) delay tasks with soft real-time constraints,

   (b) reassign priorities using the cost function if necessary,

   (c) reconfigure the DCM, if necessary,

   (d) terminate the current configuration if a task with a higher priority occurs.

This results in a pre-emptive scheduling approach, which allows the termination of a configuration. Furthermore, it uses a combination of static list scheduling and a novel dynamic scheduling approach. It considers resource constraints, such as a single ICAP, the communication infrastructure, or the maximal number of possible processors. Furthermore, it is used to schedule both hardware and software tasks. Moreover, the clock frequency of processing elements can be increased at runtime, if necessary, and the configuration times as well as the communication costs between tasks are considered. Another degree of freedom is that, while a hardware task is loaded via the ICAP interface, a software task can be loaded simultaneously via the on-chip communication infrastructure.

4.2. Resource Allocation of the CAP-OS. After the scheduling, the CAP-OS tries to allocate a resource for the *Ready* task

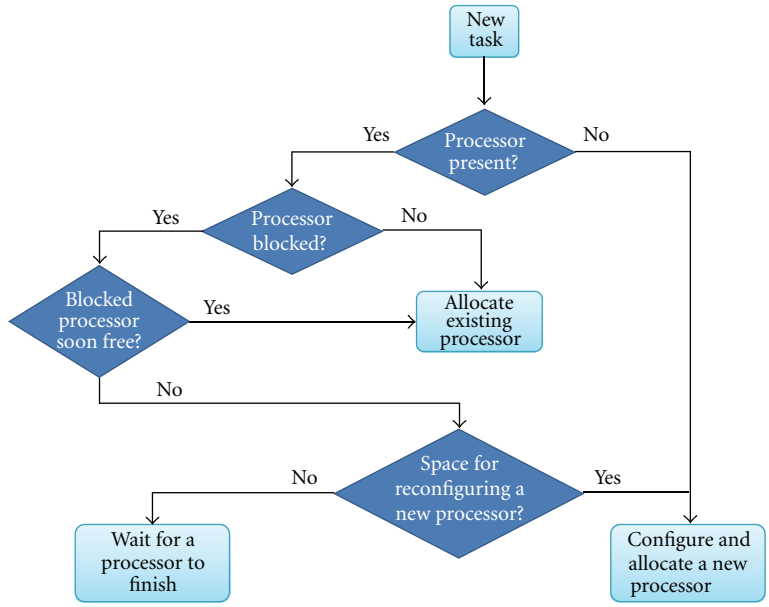


Figure 7: Decision tree for resource allocation for an SW task.

with the highest priority. For the resource allocation, the decision is made as shown in Figure 7.

First CAP-OS analyzes if a processor is present and available on the reconfigurable hardware. If no processor is present, a new one is configured and reserved for the new task. If processors are present in the system, it searches for one, which is not blocked by another task. If all existing processors are blocked, it is checked if one of them will finish its execution soon. This is important, because the reconfiguration and allocation takes time. If an existing processor finishes in a shorter amount of time than the reconfiguration time of a new processor, the reuse of this existing processor is preferred. This also has the benefit to reduce the area utilization and therefore to reduce the overall power consumption. If none of the existing processors will finish soon, it is analyzed if the maximal number of processors is reached or if there is still space to reconfigure a new processor. If there is space on the reconfigurable hardware, a new processor is reconfigured and allocated for the new task. If not, the new task has to wait until one of the processors becomes available.

The same procedure is done for a codesign task, because this task is a combination of an SW task and a hardware accelerator. Here, the available processor is extended with an accelerator, while the software part of the codesign task is loaded into the chosen processor.

Pure hardware tasks also exist. These are requests from existing RAMPSoC processors, which either need an accelerator to increase performance or which want to exchange the existing accelerator against a different one due to requests from the environment. Such exchange requests could be, for example, the exchange of an image-processing filter due to a change in the incoming frame of the image. In that case, no decision tree is required, and the requesting processor is extended by configuring the requested hardware accelerator via the ICAP interface.

4.3. Configuration Management. After the *Ready* task with the highest priority has been successfully assigned to a processor, this task is assigned to the configuration management. The configuration management is responsible for handling the configuration of the tasks via the ICAP and also for loading software tasks into already existing processors over the communication infrastructure. It is also responsible for pre-empting a current configuration if another task with a higher priority needs to be configured. As mentioned before, a terminated configuration has to restart again from the beginning, because Xilinx FPGAs do not support the continuation of a terminated configuration so far. On the other side, software tasks, under configuration via the communication infrastructure, can be interrupted if a task with higher priority occurs. Afterwards, the terminated configuration can be resumed. Furthermore, it is possible to configure a hardware and a software task or two software tasks simultaneously by using the configuration via the communication infrastructure for one of the software tasks, while the other tasks are configured via the ICAP interface. Therefore, the configuration management of the CAP-OS distinguishes between three types of configurations as shown in Table 2.

The term *soft* and *medium* means an interruptible and *hard* means a noninterruptible configuration. Soft configurations are new software tasks that get loaded via the communication infrastructure into an existing processor. As they can be pre-empted and continued easily, they can be interrupted any time if a task with high priority occurs. Medium configuration types are, for example, the configuration over the ICAP interface of software tasks or hardware accelerators for existing processors. As soon as 80% of the corresponding bitstream of a medium configuration type is configured, this element changes to be a hard configuration type. The reason is to prevent the termination of a nearly finished configuration, because the already configured data would

Table 2: Configuration types.

Configuration type	Features	Elements
Soft (communication infrastructure)	Interruptible	Software
Medium (ICAP)	Interruptible until 80% of the bitstream are reconfigured	Software, accelerator
Hard (ICAP)	Not interruptible	Processor, DCM

Table 3: Performance of the currently supported different interfaces in RAMPSoC.

Interface	Performance (100 MHz, Virtex-4)
FSL-ICAP [18]	28,28 MB/s
Point-to-point via FSLs (Fast Simplex Links) [19]	13,09 MB/s

be lost. 80% is a default parameter and can be changed by the user, depending on the application requirements. Other examples of hard configuration types are the configuration of the DCMs and of the processors, because the configuration of a DCM is urgent and fast, and the processor is far less task specific than an accelerator.

The decision to configure a software task via the ICAP interface or via the communication infrastructure depends on the mobility of the task, the availability of the interfaces, and the loading speed of the interface, which depends on the target platform and the chosen interface. Table 3 gives an overview about possible interfaces and how is their performance.

It is possible to increase the throughput of these interfaces by connecting them directly to the external memory instead of using the processor to load the data from external memory. An example is the PLB-ICAP from Claus et al. [20], which achieves a throughput of 400 MB/s. It is therefore planned to provide a direct memory access also to the FSL-ICAP to further increase its performance. The performance of communication infrastructure can be increased in a similar fashion.

4.4. Communication Establishment between Tasks. After successfully configuring a task, the CAP-OS tries to establish a communication with this task and to transfer information about the IDs of the communication partners to it. Figure 8 illustrates the required steps, to successfully establish a communication between the different tasks at runtime.

The five runtime communication establishment steps required after a task has been mapped onto a processor x are

(1) CAP-OS sends sync word to processor x;

(2) processor x responds with the same sync word to ensure a correct communication;

(3) CAP-OS sends task info (Task ID, number of predecessor/successor tasks, and their IDs) to processor x. This task info is required by the task to find its communication partners at runtime;

(4) processor x sends its Task ID to all other processors, and it checks each of its communication links for the Task ID of its communication partners. It has to send its Task ID to all other processors, because it could happen that a predecessor and a successor will be mapped onto the same processor. An example for such a case will be given in Section 6;

(5) after execution, processor x informs CAP-OS that it is now free for a new task.

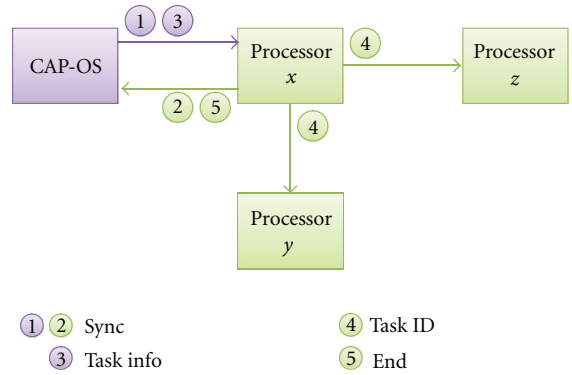


Figure 8: Runtime communication establishment steps between different tasks.

5. Integration of CAP-OS on RAMPSoC

CAP-OS software is integrated into an RAMPSoC on one of the available microprocessors. On the selected microprocessor, a state-of-the-art RTOS with multithreading capabilities is implemented. On top of this RTOS, the CAP-OS is implemented using different threads for the different functionalities. As shown in Figure 9, this microprocessor is directly connected with the Xilinx ICAP primitive via an FSL connection. Furthermore, the processor has access to an external memory, in which the partial bitstreams of the tasks are stored.

The microprocessor is connected with the other processors in this example over the star-wheels NoC. A point-to-point connection with each of the other partners or a connection over a different NoC or a bus is also supported. Several possible choices for an on-chip microprocessor exist. The IBM PowerPC 405 (PPC405) [21] was chosen for running CAP-OS. It is available on Xilinx Virtex-4FX FPGAs as a hard IP core. The main reasons for choosing the PPC405 are the support of high frequencies up to 450 MHz and the availability on the Virtex-4FX100 FPGA on the used target FPGA board from Alpha-Data [22]. High frequencies are important to execute the CAP-OS with a low latency to support and enable the real-time requirements. Other

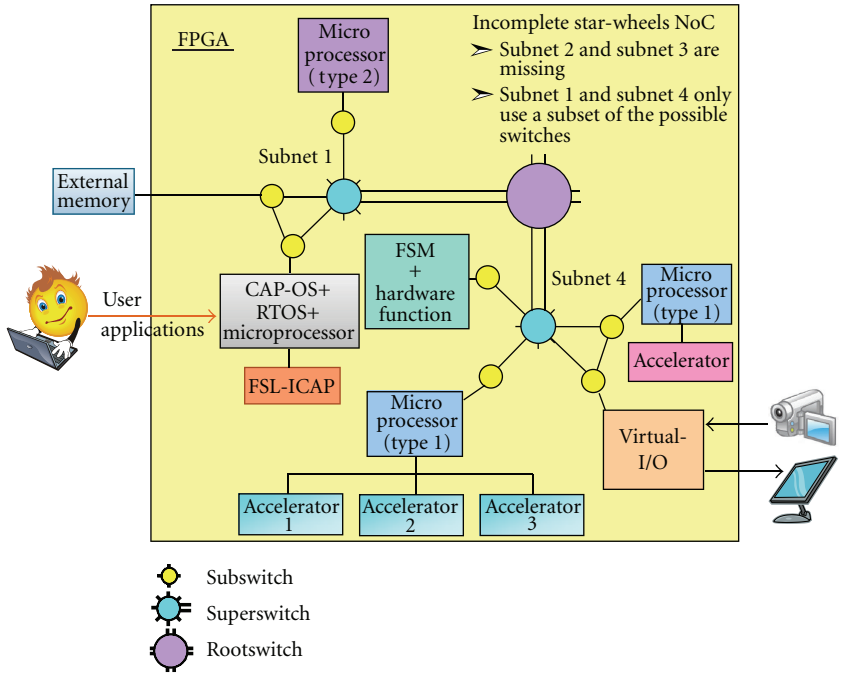


Figure 9: Integration of the CAP-OS on the RAMPSoC.

possible microprocessors would be soft IP cores, such as Xilinx MicroBlaze or Leon SPARC, but they lack with the support of such high frequencies. As the new Xilinx FPGAs, such as Virtex-6, does not provide the PowerPC anymore, an alternative version of CAP-OS for the MicroBlaze processor was also realized and implemented.

After selecting the processor, an appropriate RTOS was chosen. The demands for the RTOS are

(i) proven support of PPC405 and MicroBlaze,

(ii) multithreading capabilities,

(iii) small memory footprint.

Several different RTOSes exist, but due to the reasons above, the Xilkernel [23] from Xilinx was selected. The CAP-OS is programmed in C, and its functionalities are implemented in several different threads, which are executed in Xilkernel using multithreading. For scheduling the different threads, Xilkernel offers two scheduling policies: round robin or priority-based scheduling. Priority-based scheduling was chosen to execute the different CAP-OS threads according to their priorities.

Furthermore, the processor is directly connected to the ICAP primitive and to an external memory (DDR2 SDRAM), in which the bitstreams are stored. The CAP-OS and Xilkernel are executed using on-chip memory for maximum performance. In the following subsection, the implementation of the different CAP-OS threads is described in detail.

5.1. Implementation of the CAP-OS. The CAP-OS is programmed using six threads as shown in Table 4.

The priorities are sorted with increasing numbers starting with the highest priority from 0. Test_main is the startup thread and has a fixed priority. The priorities of the other five threads can change at runtime depending on the demands of the applications. The three threads with priority level 3 (Schedule, Configure, and Contr_Exit_Task) compete against each other, after the first three threads with higher priority have finished executing. While the other threads only execute in the beginning once, these three concurring threads execute until the last task finishes executing.

6. Implementation and First Results

The functionality of the CAP-OS as specified was evaluated by implementing an RAMPSoC system on the target Alpha-Data FPGA board. The CAP-OS was implemented using one of the available PPC405s and the Xilkernel RTOS. The maximum number of reconfigurable processors was set to four, to be artificially below the number of tasks within our evaluation task graphs. As the target Virtex-4FX 100 FPGA provides a large number of reconfigurable resources, a higher number of processors could be used, if necessary. As reconfigurable processor, the Xilinx MicroBlaze (μBlaze) [9] was chosen due to its small area footprint and the compatibility to the PPC405. As shown in Figure 10, the Fast Simplex Links (FSLs) [19] are utilized for the communication between the processors. The decision for these communication infrastructures has its basis in the fact that FSLs offer an FIFO-based unidirectional communication and that for the limited number of processors, an NoC would create a high overhead in terms of utilized area. The PPC405 can be connected via FSL to 32 data sinks and sources, while each μBlaze could be connected to 16.

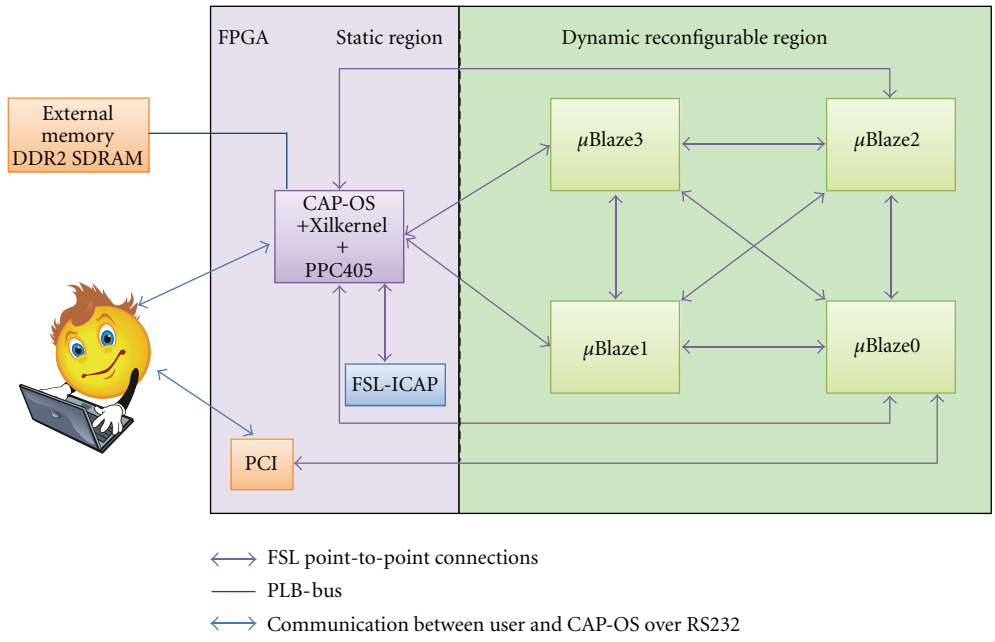


Figure 10: Implemented RAMPSoC system.

Table 4: Realized threads of the CAP-OS.

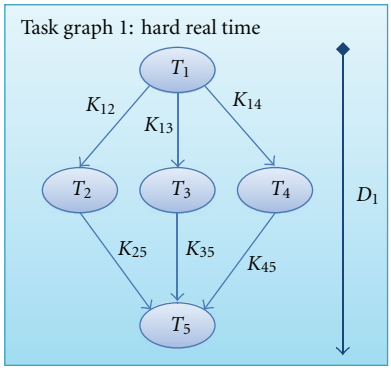
Thread	Priority	Description
Test_main	0	Initial thread. Launches the other five threads.
Init_proc	1	Generates a list containing all possible processors and their attributes. Executes only once.
Task_graph	2	Initialization of the tasks and generation of the task graphs. Calculation of ALAP and ASAP start time and the mobility of each task. Matching of tasks with equal requirements (HW constraints, same algorithm)
Schedule	3	Scheduling of the *Ready* tasks and processor allocation.
Configure	3	Configuration management for the scheduled and allocated task and communication establishment between the new configured task and its neighbors.
Contr_Exit_Task	3	Controls the executing tasks. If a task finishes execution, the occupied processing element is freed.

Additionally, the FSL-ICAP IP core from Xilinx together with an external DDR2 SDRAM is connected to the PPC405. The user interface to the CAP-OS is realized through an RS232 port. For the preliminary tests, the dynamic and partial reconfiguration was not deployed, because the scope was to verify the CAP-OS and not the FSL-ICAP primitive. Instead of sending the partial bitstreams to the ICAP core, a counter within the *Configure* thread was used, to simulate artificially the reconfiguration times of the different tasks. For reconfiguring a whole processor via the ICAP interface 5 ms, and for loading a software task onto an existing processor via the FSL communication infrastructure, 2 ms were assumed. These times are-worst case scenarios, and certainly the reconfiguration time and the loading time vary in real scenarios depending on the size of the bitstream/the software executable. Table 5 shows the estimated reconfiguration times for an average size MicroBlaze and for the size of a typical image processing application software.

At system startup, it is assumed that only the static part is present and the other processors will be "reconfigured" on demand. Physically, the system shown in Figure 10 was present from the beginning, and after the artificial simulation of the reconfiguration time is finished, the corresponding processor is activated. For the verification of the CAP-OS functionality and to measure the timing overhead of the current CAP-OS implementation, TG1 shown in Figure 11 was used. TG1 has hard real-time constraints. This could be for example, an image-processing application, which receives the images from a camera and has to provide the results of an image-processing algorithm to the user via a monitor in real time. Therefore, the global deadline (D_1) of TG1 is 40 ms

Table 5: Reconfiguration/loading times for the chosen interfaces: FSL-ICAP and FSL point-to-point links.

Type	Size	Reconfiguration/loading time
Reconfiguring a MicroBlaze	ca. 120 KB	4,24 ms
Loading a software	16 KB	1,22 ms



Task	Description	Algo-ID
T_1, T_5	Software tasks with same hardware requirements: for example, processor with PCI connection	0
T_2, T_3, T_4	Software tasks, for example, different image processing filter	1, 2, 3

Figure 11: Task graph used for the CAP-OS evaluation: $D_1 = 40$ ms.

Table 6: Timing overhead of CAP-OS for processing TG1.

Thread	Average number of clock cycles per call
Init_proc	2118
Task_graph	9022
Schedule	650
Contr_Exit_Task	227

using a camera with a frame rate of 25 Hz. If this deadline is missed, frames will be lost.

To measure the timing overhead, the CAP-OS was executed on the FPGA using TG1. To proof, if the CAP-OS correctly reuses existing resources, the two tasks $T1$ and $T5$ were set to have the same algorithm (same Algo-ID) as shown in Figure 11. During the execution on the FPGA, the number of clock cycles, required per call by each thread, were measured. The results for the timing overhead provided by the CAP-OS are shown in Table 6.

The clock cycles of the *Configure* thread depend on the size of the bitstream and on the speed of the ICAP primitive. Therefore, they are not explicitly presented here. *Test_main* only launches the other five threads, but itself does not produce timing overhead and is therefore also not mentioned here. Of course, *Init_proc* depends on the number of processors (here four), and *Task_graph* depend on the TG (here TG1 with five tasks). Therefore, these numbers are just an example for the given TG. The clock cycles required for the *Schedule* thread depends on the complexity of the scheduling. For example, they increase slightly if the cost function needs to be evaluated for two tasks. *Contr_Exit_Task* is very stable.

With this example, it can be shown that CAP-OS worked correctly as specified and assigned the tasks of TG1 without violating the global deadline. Also, the resource reuse worked correctly. $T5$ was allocated onto the same processor as $T1$, because they have the same algorithm, and this way the reconfiguration time could be saved.

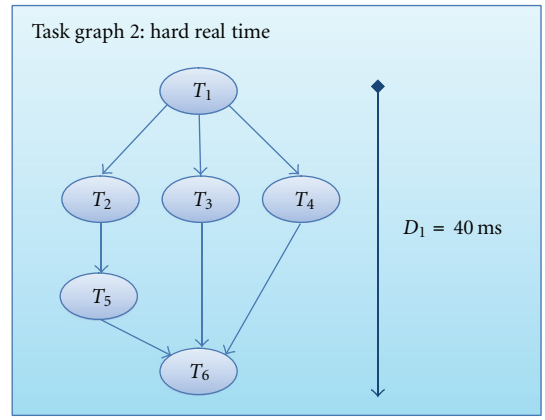


Figure 12: Task graph TG2 of an image-processing application, which detects template-based and point-like objects within an image.

7. Case Study with Image-Processing Scenario

Finally, a case study using an image-processing application for object recognition was designed. Figure 12 illustrates the task graph of the application. Task 1 is a pure software task and receives an input image via PCI bus. It then forwards the complete image to task 2. After partitioning the image into two overlapping tiles, it forwards the upper half of the image to task 3 and the lower half to task 4. Task 2 equalizes the histogram of the image. As this is a very compute intensive task, parts of the algorithm are outsourced in an accelerator. Therefore, task 2 requires both a processor for the software part of the algorithm and a closely coupled accelerator for the compute intensive part of the algorithm. Task 2 sends its results to task 5. Task 5 tries to find objects within the equalized image by comparing a predefined template with the input image using the SAD (sum of absolute differences) algorithm. It is implemented in software and forwards its results to task 6. Task 3 and task 4 execute both the hotspot detector algorithm in software on different parts of the image. The hotspot detector algorithm is an image-processing algorithm, which searches inside an image for bright point-like objects. The results of task 3 and task 4 are then forwarded as well to task 6, which is responsible for collecting all results and forwarding them to the Host PC via the PCI connection. Like task 1, task 6 is also implemented as a software task, and both require a processor with a PCI connection.

To measure the execution times of each task separately, each task was implemented on a single MicroBlaze processor running at 100 MHz on the target FPGA platform. For the measurement, an input image with the size of 64×64 pixels was used. An exception is task 2. The execution time of this task was measured using a single MicroBlaze connected via FSL with the hardware accelerator. The size of the software executable file for each algorithm is received using the GCC compiler within the Xilinx Platform Studio [24]. This size is important for calculating the loading time via the FSL communication infrastructure into the local memory of a processor on the FPGA.

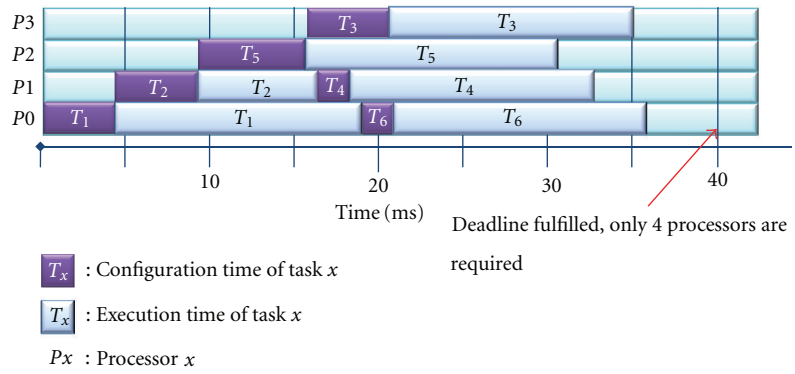


Figure 13: Theoretical results of CAP-OS for the image-processing applications and respecting heterogeneous configuration times.

For each design, the place and route report was used to extract the values for the utilized area, this means the required number of CLBs, block RAM (BRAM), and DSPs for the MicroBlaze processor. For task 2, the amount of resources for the accelerator was also taken into account.

The smallest addressable segment of a Virtex-4 FPGA configuration memory space is called a frame and covers a height of 16 CLBs or 4 BRAMs or 8 DSPs. It has a length of 1312 bits [25]. 22 frames are needed to reconfigure a column of 16 CLBs. To reconfigure 4 BRAMs, 64 frames are needed for the BRAM content, and 20 frames for the BRAM interconnect. To reconfigure 8 DSPs, also 21 frames are needed. Therefore, the number of frames required for the partial bitstream is calculated as shown in the following Formula:

$$\#\text{Frames} > \left\lceil \frac{\#\text{CLBs}}{16} \right\rceil * 22 + \left\lceil \frac{\#\text{DSPs}}{8} \right\rceil * 21 + \left\lceil \frac{\#\text{BRAMs}}{4} \right\rceil * (64 + 20). \tag{8}$$

The reconfiguration time is then calculated by using this Formula:

$$\text{Reconfiguration Time} = \frac{\#\text{Frames} * \text{Framelength}}{8 * \text{Throughput FSL\_ICAP}}. \tag{9}$$

In Table 7, a detailed description for each task of TG2 together with the measured execution times and the estimated reconfiguration times are presented. The reconfiguration times are worst-case reconfiguration times. This means that for each task the reconfiguration of a new processor has been assumed. These reconfiguration times are the input for the static list scheduling algorithm of the CAP-OS described in Section 4. The static list scheduling algorithm is used to assign each task to a priority, based on the mobility of the task. The mobility of a task is the subtraction of its ASAP start time from its ALAP start time as shown in Formula (3). Table 7 shows the resulting priorities for TG2.

Furthermore, the last column in Table 7 shows the Algo-ID of each task. The Algo-ID is used by the dynamic scheduling decision of CAP-OS to decide which task may reuse an existing processor. $T3$ and $T4$ have the same Algo ID, because they execute the same software algorithm on a different data set. $T1$ and $T6$, which both execute a different algorithm, have the same Algo-ID, because they have a common hardware restriction, as they require both a processor with a connection to the PCI interface.

Figure 13 shows the calculated results of CAP-OS for the TG2. As can be seen, four processors $P0$ to $P3$ were used. Figure 10 shows how the final implemented system would look like, where P0 is the only processor with a PCI connection. Due to this, that processor is reused by task 6 after task 1 has finished. Task 3 and task 4 are mapped onto different processors even though they have the same Algo-ID, because otherwise the global deadline would be violated. Also, the benefit of using the FSL communication infrastructure for loading software into an existing processors and therefore keeping the ICAP interface available for the hardware reconfiguration can be seen. For example, processor P3 is being reconfigured via the ICAP interface while simultaneously the software executable of task 4 is loaded over the FSL communication infrastructure to processor P1.

Table 8 shows the estimated execution times for mapping the application sequentially on one processor in the first row, parallelizing it using for each task a processor in the second row and the solution proposed by CAP-OS, which uses four processors in the last row. Furthermore, Table 8 shows the part of the execution time, which is spent for reconfiguring the hardware and for loading the software.

As can be seen, both the uni-processor design and the 6-processor designs are static and do not require any reconfiguration time.

The uni-processor design is the slowest solution and violates the global deadline of the application, because it can only execute the complete application sequentially.

The 6-processor design is the fastest solution, but also has the highest resource requirements and therefore the highest power consumption. Furthermore, some of the processors are idle (e.g., processor with task 6), while others are executing (e.g., task 3, 4, and 5). This results in a bad workload balancing.

Table 7: Detailed information about each task of the TG 2 at 100 MHz.

Task	Description	Rec. time in ms	Exe. time in ms	Priority	Algo ID
$T1$	SW Task: Read from PCI	4,24	1	1	1
$T2$	SW Task + Accelerator: Histogram equalization	4,5	2,25	2	2
$T3$, $T4$	SW Task: Hotspot detector	4,24	14,5	4,5	3
$T5$	SW Task: SAD (sum of absolute differences)	4,24	17,25	3	4
$T6$	SW Task: Send to PCI	4,24	1	6	1

Table 8: Estimated execution times for the TG2.

System	Estimated execution time in ms	Time spent for HW reconfiguration/SW loading time
UniProcessor	50,5	0
6 Processors	17,25	0
CAP-OS (4 Processors)	31,72	17,22/2,44

CAP-OS with its four processors provides a meet-in-the middle solution by reusing existing processors. It therefore achieves a good balance between performance and power consumption. It is faster than the uni-processor design and fulfils the global deadline of the application. Moreover, it requires fewer processors than the 6-processor solution and therefore has lower power consumption. Also, as can be seen in Figure 13, the workload between the processors is well balanced. It can be further seen that as soon as a processor finishes its current task, it is allocated for executing the next task of the application.

It has to be mentioned here that the execution times of the different tasks, for example, $T1$, are longer in this figure than the ones given in Table 7. The reason for this is the consideration of the communication between the tasks. For example, task 1 has to wait until all its successors are reconfigured, before it can finish its execution.

8. Conclusions and Outlook

In this paper, the concept and the features of a special-purpose OS called CAP-OS were presented. The CAP-OS is responsible for the scheduling, the resource allocation, and reconfiguration and for managing the access to the configuration access port. The CAP-OS has been integrated into the RAMPSoC approach to handle the runtime organization for the adaptive RAMPSoC hardware architecture. The CAP-OS was implemented using six threads on the Xilkernel RTOS running on a PPC405 and on a MicroBlaze processor. The correct functionality and the timing overheads of the CAP-OS were measured on the FPGA using an example task graph. The benefits of the CAP-OS were shown using a case study with a task graph from an image-processing application and comparing the results against both a uni-processor design and a complete parallel design.

Future work will be the extension of the CAP-OS to support the reconfiguration of the communication infrastructure. Furthermore, it will be extended to handle not only the demands of the user, but also the reconfiguration demands of the other processors within the RAMPSoC. These demands are mainly the reconfiguration of the accelerators if at runtime, for example, a different accelerator is required depending on the currently processed data. Furthermore, the CAP-OS will be further evaluated and will be also tested using real dynamic and partial reconfiguration. The implementation of a partial reconfigurable design of the presented case study is currently under development to evaluate that the calculated results are equal to the statically measured ones. Additional extensions of CAP-OS will be the support of merging several bitstreams and supporting bitstream relocation. Bitstream relocation is important to reduce the amount of required external memory for storing each bitstream for each possible location.

References

[1] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, and J. Weglarz, *Scheduling Computer and Manufacturing Processes*, Springer, Berlin, Germany, 2001.

[2] D. Göhringer and J. Becker, "High performance reconfigurable multi-processor-based computing on FPGAs," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW '10)*, pp. 1–4, Atlanta, Ga, USA, April 2010.

[3] P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu, "An overview of reconfigurable hardware in embedded systems," *EURASIP Journal on Embedded Systems*, vol. 2006, Article ID 56320, 19 pages, 2006.

[4] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '07)*, pp. 123–128, April 2007.

[5] M. Ullmann, M. Hübner, B. Grimm, and J. Becker, "On-demand FPGA run-time system for dynamical reconfiguration with adaptive priorities," in *Proceedings of the International Conference on Field Programmable Logic and Application (FPL '04)*, pp. 454–463, August 2004.

[6] E. Lübbers and M. Platzner, "ReconOS: an RTOS supporting hard- and software threads," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 441–446, August 2007.

[7] J.-Y. Mignolet, S. Vernalde, D. Verkest, and R. Lauwereins, "Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances," in *Proceedings of the International Conference on Engineering Reconfigurable Systems and Architecture*, 2002.

[8] D. Göhringer, B. Liu, M. Hübner, and J. Becker, "Star-wheels network-on-chip featuring a self-adaptive mixed topology and a synergy of a circuit- and a packet-switching communication

protocol," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '09)*, Prague, Czech Republic, September 2009.

[9] "MicroBlaze Processor Reference Guide, Embedded Development Kit, EDK 9.2i," UG081 (v8.1)., http://www.xilinx.com/.

[10] "Leon Sparc," http://www.gaisler.com/.

[11] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Application (FPL '06)*, pp. 1–6, Madrid, Spain, August, 2006.

[12] O. Sander, L. Braun, M. Hübner, and J. Becker, "Data reallocation by exploiting FPGA configuration mechanisms," in *Proceedings of the 4th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications (ARC '08)*, vol. 4943 of *Lecture Notes in Computer Science*, pp. 312–317, London, UK, March 2008.

[13] D. Göhringer, J. Obie, M. Hübner, and J. Becker, "Impact of task distribution, processor configurations and dynamic clock frequency scaling on the power consumption of FPGA-based multiprocessors," in *Proceedings of the 5th International Workshop on Reconfigurable Communication Centric Systems-on-Chip (ReCoSoC '10)*, KIT Scientific Publishing, Karlsruhe, Germany, 2010.

[14] "MPI: A Message-Passing Interface Standard, Version 2.2," Message Passing Interface Forum, September 2009, http://www.mpi-forum.org/.

[15] D. Göhringer, M. Hübner, M. Benz, and J. Becker, "A design methodology for application partitioning and architecture development of reconfigurable multiprocessor systems-on-chip," in *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '10)*, pp. 259–262, Charlotte, NC, USA, May 2010.

[16] "Virtex-4 FPGA Configuration User Guide," UG071 (v1.11), June 2009, http://www.xilinx.com/.

[17] D. Göhringer, J. Obie, A. Braga, M. Hübner, C. Llanos, and J. Becker, "Exploration of power-performance tradeoffs through parameterization of FPGA-based multiprocessor systems," in *Proceedings of the the 5th International Workshop on Reconfigurable Communication Centric Systems-on-Chip (ReCoSoC '10)*.

[18] M. Hübner, D. Göhringer, J. Noguera, and J. Becker, "Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs," in *Proceedings of the Reconfigurable Architectures Workshop (RAW '10)*, Atlanta, Ga, USA, April, 2010.

[19] "Fast Simplex Link (FSL) Bus (v2.11a)," DS449, June 2007, http://www.xilinx.com/.

[20] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hübner, and J. Becker, "A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, Heidelberg, Germany, September 2008.

[21] "PowerPC Processor Reference Guide," UG011 (v.1.2), January 2007, http://www.xilinx.com/.

[22] "Alpha Data," http://www.alpha-data.com/.

[23] "Xilkernel v3_00_a," EDK 9.1i, December 2006, http://www.xilinx.com/.

[24] "Embedded System Tools Reference Manual, Embedded Development Kit, EDK 9.2i," UG111 (v9.2i), September 2007, http://www.xilinx.com/.

[25] "Virtex-4 Configuration Guide," UG071 (v1.5), January 2007, http://www.xilinx.com/.