Information and Communication Technologies (ICT) Programme

Project N^o: FP7-ICT-287804

D4.3: Configuration Scheduler Requirements and Basic Functionality

Author(s): Kyprianos Papadimitriou (FORTH)
Dionisios Pnevmatikatos (FORTH)
Catalin Ciobanu (CHT)
Georgi Gaydadjiev (CHT)
Iakovos Mavroidis (FORTH)

Status -Version: Version 1.4 (Final)

Date: September 10, 2012

Distribution - Confidentiality: Public

Code: FASTER_D4_3_FOR_FF-20120910.docx

Abstract:

The run-time system manager (RTSM) is responsible for the on-line scheduling of partially reconfigurable tasks and the device management. In this document we discuss the requirements of the runtime configuration scheduler along with its basic functionality. We also identify the inputs that are necessary at compile time and during runtime needed in order to take decisions at runtime, and finally the type of operations controlled by the RTSM.

Disclaimer

The FASTER Consortium is the following:

Beneficiary Number	**Beneficiary name**	**Beneficiary short name**	**Country**
1(coordinator)	Foundation for Research and Technology – Hellas	FOR	Greece
2	Chalmers University of Technology	CHT	Sweden
3	Imperial College London	IMP	UK
4	Politecnico di Milano	PDM	Italy
5	Ghent University	GNT	Belgium
6	Maxeler	MAX	U.K.
7	ST Microelectronics	STM	Italy
8	Synelixis	SYN	Greece

Document Revision History

Date	**Issue**	**Author/Editor/Contributor**	**Summary of main changes**
June 18, 2012	0.1	Kyprianos Papadimitriou	Initial draft
July 19, 2012	1.1	Kyprianos Papadimitriou	Changes after suggestions by Marco Santambrogio and Georgi Gaydadjiev for Standalone and OS implementations of the RTSM; Extended text.
August 10, 2012	1.2	Kyprianos Papadimitriou, Dionisios Pnevmatikatos, Iakovos Mavroidis	Addressed suggestions by Christian Pilato and Marco Santambrogio
September 3, 2012	1.3	Karel Brunel, Kyprianos Papadimitriou	Micro-reconfiguration support; refinements; added the last sections
September 9, 2012	1.4	Catalin Ciobanu	Quality Control

Table of contents

1. Introduction

The run-time system manager (RTSM) is responsible for the on-line scheduling of partially reconfigurable tasks and the device management. This deliverable discusses:

- The way the run-time system reads and manages the directives and the task graph provided by the compilation/scheduling stage. The RTSM should provide a solution according to this information;
- Whether the run-time system software code will be part of the OS itself, or run above the OS;
- The decisions made by the RTSM. The main ones will be: i) the time slot in which the reconfiguration of a module will occur; ii) the portion of the FGPA on which the module is going to be placed and iii) the time slot in which its execution will start;
- The way the scheduler processes the data from a dependency/communication graph;
- The criteria based on which the RTSM makes decisions. Provisionally, these criteria are: i) reconfiguration time; ii) device area constraints; iii) precedence between the modules and iv) fragmentation level;
- The way the caching and prefetching of configuration bitstreams techniques will be incorporated;
- Whether thermal and power values (called non-functional data), will be used as feedback to the RTSM for taking decisions such as task rescheduling and task unloading;
- The level of transparency to the user;
- The integration of micro-reconfiguration into the RTSM.

The result of the present deliverable will be the exact definition of the distinct scheduler components, the input data needed from compile time and during runtime needed to take decisions at runtime, and finally the type of operations controlled by the RTSM. The present deliverable is connected with D4.1 of WP4 (Deliverable_4.1, Feb 2012) , D1.2 of WP1 (Deliverable_1.2, May 2012), D4.2 of WP4 (Deliverable_2.2, Aug 2012) and D2.2 of WP2 (Deliverable_4.2, Aug 2012).

2. Requirements of the Runtime Scheduler

In order for the runtime scheduler to operate properly, a set of distinct components are required. Also, certain inputs will drive the scheduler to decide the correct functionality. The scheduler determines which task is going to run on which platform block; a block can be a discrete component such as a CPU, a static hardware part, or, a partially reconfigurable part of an FPGA. Task scheduling, i.e. which task is going to run on which platform block – CPU or hardware block, will be conducted at run-time and governed by a run-time system manager (RTSM) running in a CPU.

The run-time scheduler communicates with the baseline scheduler, which is provided initially with the task graph representing the task dependencies, and initially mapped in the basic architecture. The communication interfaces between the components of the platform, e.g. processors, external memory, are determined at design time, and the drivers (these drivers are fixed and cannot be changed at run time) needed by the processor to invoke and manage each core are generated. The runtime system is mainly responsible to manage the core execution, reconfiguration calls and the interrupt routines associated with each core.

2.1. Scope of the scheduler

The main components of the RTSM are the Placer (P), Scheduler (S), Translator (T), Loader (L), described in (Deliverable_4.1, Feb 2012), (Deliverable_1.2, May 2012). These distinct components cooperate with each other in order to control the loading and execution of tasks. The Scheduler cooperates with the Placer for finding the proper processing element (PE), such as a partially reconfigurable area or a software component, in order to execute a task. The scheduler is responsible for operations such as finding the time slot in which a task is loaded (for a partially reconfigurable region this means that a reconfiguration should occur) and the task execution starting time. Also, in case a request for loading a HW task to the reconfigurable logic is made but there is no available space, the scheduler should either activate a SW version of the task (if it is available), or schedule the task for future execution based on predicted free space and specific starting times. The scheduling policy (e.g. Round Robin, First Come First Served, Best Fit, Earliest Deadline First, etc.) can be accessed as a separate library.

2.2. Scheduler operation and input needed

In order to provide the proper inputs to the run-time scheduler, we explore which parameters are determined at compile time but remain unchanged during execution, and which are changed during execution. The following analysis explains the way the scheduler will operate and identifies the input needed to feed the scheduler.

All tasks can have SW and HW versions. The HW tasks are predesigned, i.e. synthesized at compile time, and stored as partial bitstreams in a repository, according to the restrictions of the particular FPGA technology used (e.g., Xilinx). Each HW task is characterized by three parameters: *task area (width and height), reconfiguration time, and execution time* (Marconi, 2011).

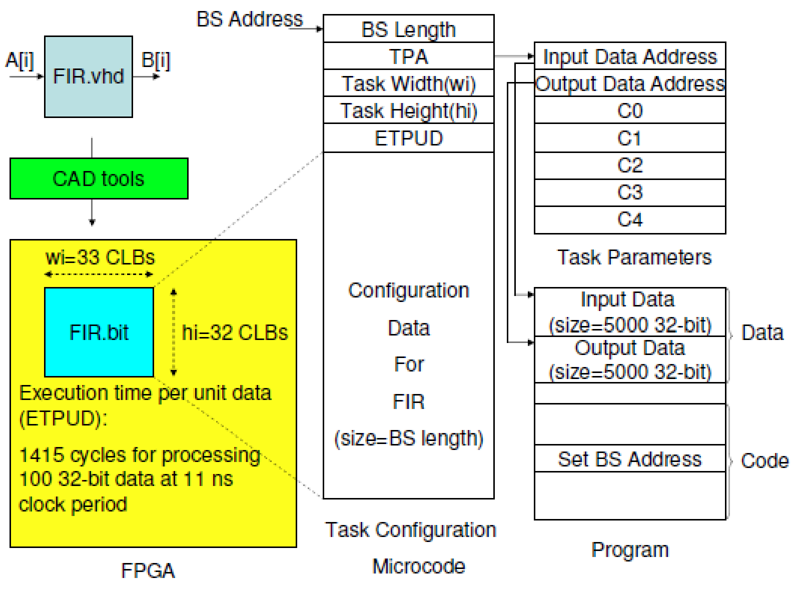
A[i]
FIR.vhd
B[i]
CAD tools
wi=33 CLBs
FIR.bit
hi=32 CLBs
Execution time per unit data (ETPUD):
1415 cycles for processing 100 32-bit data at 11 ns clock period
FPGA
BS Address
BS Length
TPA
Task Width(wi)
Task Height(hi)
ETPUD
Configuration Data For FIR (size=BS length)
Task Configuration Microcode
Input Data Address
Output Data Address
C0
C1
C2
C3
C4
Task Parameters
Input Data (size=5000 32-bit)
Output Data (size=5000 32-bit)
Data
Set BS Address
Code
Program

Figure 1 System at Design Time

Figure 1 illustrates the detailed operation of task creation at design time. The configuration data and task specific information are merged together in a so-called Task Configuration Microcode (TCM) block. TCM is stored in the memory at the Bitstream (BS) address. The BS field corresponds to the size of the configuration data field. The Task Parameter Address (TPA) defines where the task input/output parameters are located. Task Width and Height hold the size of the task, while Execution Time Per Unit of Data (ETPUD) has the task execution time per a specific amount of data.

In the example of Figure 1, the HW task is a simple Finite Impulse Response (FIR) filter. The task consumes input data from array A[i] and produces output data stored in B[i], where $B[i] = C0*A[i] + C1*A[i+1] + C2*A[i+2] + C3*A[i+3] + C4*A[i+4]$ and all data elements are 32 bits wide. The task implemented in hardware description language (HDL) (FIR.vhd) is synthesized by commercial CAD tools that produce the partial bitstream file (FIR.bit) along with the additional synthesis results for that task. The bitstream contains the configuration data that should be loaded into the configuration memory to instantiate the task at a certain location on the FPGA fabric. Synthesis results, or even better results from the stage in which the creation of reconfigurable areas is performed, are used to determine the rectangular area consumed by the task, in terms of configurable logic blocks (CLBs), specified by the width and the height of the task. In our example, the task width is 33 and the task height 32 CLBs for Xilinx Virtex technology.

In a realistic scenario, one additional design space exploration step can be added to steer task shapes towards an optimal point, a concept that will be studied within the context of FASTER project. In particular, in a stage of the FASTER tool flow, both task sizes and reconfiguration times are predicted using high-level models; these values are included in the XML file. A set of parameters should be taken into account in order to predict the reconfiguration time of a certain reconfigurable area: bitstream size, throughput of reconfiguration port, the characteristics of the memory used for storing the bitstream and of the reconfiguration controller.

Next, the task should be tested by the designer to determine how fast the input data can be processed. For the example of Figure 1, the FIR task needs 1415 cycles to process 100, 32-bit input data elements at 11 ns clock period making its ETPUD 1415*11 = 15565 ns per 100 32-bit unit data. Based on this ETPUD number, we can compute the task execution time for any input data size. In our example, there are 5000 32-bit input data elements that have to be processed by the FIR HW task. Therefore, the expected execution time of the FIR task is (5000÷100)*15565 = 778250 ns.

The configuration data and task specific information are merged together in the Task Configuration Microcode (TCM) block as shown in the middle of Figure 1, which is pre-stored in memory at the Bitstream (BS) Address. The first field (BS length) represents the size of the configuration data field. This value is used by the Loader when the task is fetched from memory. The Task Parameter Address (TPA) defines where the task input/output parameters are located; this is done using pointers to these locations. In Figure 1, the task parameters are the input and output data locations and the FIR filter coefficients (C0-C4). The input data address represents the location of the data which will be processed. The location where the output data should be stored is defined by the output data address.

During runtime, whenever the system needs to execute a hardware task, the Placer (P) is invoked to find a location for it on the FPGA. This is demonstrated in Figure 2. From the TCM, the Placer gets the task properties: task width, task height, reconfiguration time, and its execution time. The Placer searches for the best location for the task based on the *current state of the used area model*. When the location is found (in this example: *r'* and *c'*), the task is placed in the area model and the area model state is updated as shown on the left side of the figure. The Translator converts this location into real physical location on the targeted FPGA. In the bitstream file (FIR.bit), there is information about the FPGA location where the HW task was pre-designed (in Figure 2: *r* and *c*). By modifying this information at runtime, the Loader partially reconfigures the FPGA through its configuration port (e.g. ICAP, JTAG or SelectMAP in Xilinx devices), by using the technology specific commands at the location obtained from the Translator (*r''* and *c''* in this example). By decoupling our area model from the fine-grain details of the physical FPGA fabric, we define an FPGA technology independent environment where different FPGA vendors (e.g., Xilinx, Altera, etc.) can provide their consumers with full access to partial reconfigurable resources without exposing all of the proprietary details of the underlying bitstream formats. Also, this allows for the reconfigurable system designers to focus on partial reconfiguration algorithms without having to bother with all low-level details of a particular technology.

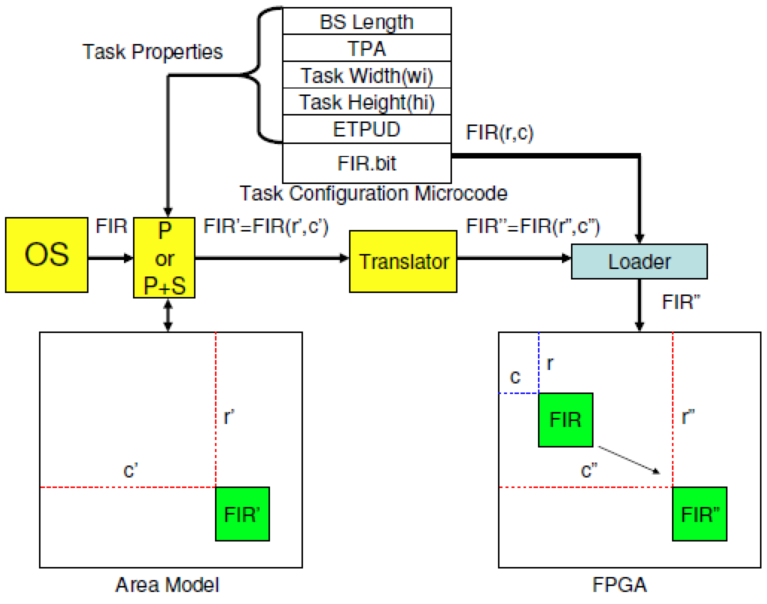

Figure 2 System at Run-time

From the above description we identify the parameters that are determined at compile time and are not changed at runtime, as well as the parameters that are changed at runtime. The former ones called static parameters should be saved in storage means that will be accessed during execution only for reading, while the latter ones, called dynamic parameters, will be saved in storage locations that will updated at runtime.

2.2.1. *Static parameters*

- Reconfigurable regions sizes: designated at compile time using the region-based method;
- Reconfiguration time: an attribute related directly with the size of reconfigurable region. Other factors contributing to the reconfiguration process already mentioned earlier are: throughput of reconfiguration port, memory storing the bitstream, reconfiguration controller; the latter component can be dedicated exclusively to the reconfiguration process, in order to avoid variations in the reconfiguration time;
- ETPUD (Execution Time Per Unit of Data): it is fixed, as it concerns the time elapsed to process a certain amount of data, affecting the task overall execution time. However, the execution time is also influenced by the overall size of the data to be processed, which might not be fixed. The execution time can also be data-dependent. An analysis on this is given in a following section;
- Tasks that at compile time are assigned to be executed in fixed Processing Elements (PE), i.e., CPU or static HW part, and tasks assigned to certain reconfigurable areas.

2.2.2. *Dynamic parameters*

- The current FPGA status (updated continuously during runtime) with regard to which reconfigurable region accommodates which task;

- The current status of each reconfigurable region. Possible conditions include: empty, (re)configuring, busy executing, not empty but idle, the ID of the task placed on the reconfigurable region;
- The current status of each task. Possible conditions include: already placed in a reconfigurable area and ready for execution, (re)configuring, running, to be executed in a partially reconfigurable area but not yet configured, the reconfigurable area in which the task is loaded.
- Task execution time. It can depend on: i) the amount of data to be processed, such as real-time data entering the system through a network link, or, ii) the nature of data to be processed. Task deadlines may be updated during runtime by monitoring the performance of each task (and thus its implementation).

2.3. Task execution time

The task execution time raises an issue that needs to be studied separately. Although the FASTER project aims at building a framework targeting four (4) applications, it should be flexible enough to cover an even broader range of applications, in which execution time of tasks can be either data dependent or independent.

It was mentioned earlier that the task execution time is specified by the time needed to process a unit of data (referred to as Execution Time Per Unit of Data or ETPUD) and the overall data size to be processed (i.e. how much data need to be processed). However, for some applications, the task execution time is also dependent on the exact data content (e.g., as in the case of Viterbi and Context-Adaptive Binary Arithmetic Coding (CABAC)). In such applications, even when processing the same amount of data, the elapsed time will be different when the input data content changes. To address data-dependent task execution times two solutions have been considered: *worst case execution time* scenario and *notification on task completion* (Marconi, 2011). In our first prototype, we assume the worst-case execution time scenario in which we use the task execution time when processing the worst-case input data content. In such scenario, it is possible that the actual task completion will happen earlier than the scheduled completion time. In addition, such earlier finished tasks may cause additional wasted area that cannot be utilized by other tasks. In such a scenario, however, the overall system will operate correctly. This scenario is the worst case with respect to the placement and scheduling algorithms due to the introduced overheads in execution time and wasted area. The second solution, notification on task completion, requires dedicated hardware support for feedback signaling when the running tasks complete, and can additionally improve the overall system performance. Some existing systems already have the necessary ingredients required to implement such support. In particular, in the MOLEN architecture, on which the FASTER Instruction Set Architecture (ISA) extensions are based on (Deliverable_4.2, Aug 2012), the sequencer is aware of the hardware (HW) task start and completion times. The only necessary extension is to provide a way to pass this information to the HW scheduler and make it aware of running tasks completion. With this knowledge, the HW scheduler can make data content dependent tasks scheduling more efficient.

2.4. Formulation of input for the runtime scheduler

The input to the runtime scheduler is provided by the directives of OpenMP by annotating the C code (terms “directive” and “pragma” are used interchangeably in the present deliverable). Two types of directives are available: one for defining the parallel sections (this is done using OpenMP pragmas) and the other one for identifying the processing element (PE) in which a task can execute (this is done using Mapping pragmas). In particular, five (5) Mapping pragmas are used for annotating the source code in order to provide the main directions to the runtime scheduler, which essentially constitute the baseline scheduler. More details are provided in (Gianluca Durelli, Christian Pilato, Andrea Cazzaniga, Donatella Sciuto, Marco D. Santambrogio, July 2012):

- Application name: it should have a unique name;
- Task name: defines a task in the application by annotating a function of the code;
- Task mapping: specifies the list of processing element (PE) to which the execution of certain tasks can be assigned at runtime. The processing elements are listed in the XML describing the architecture (Deliverable_2.2, Aug 2012). Each of these assignments defines an implementation for a task. A task can be executed in different PEs; this assignment is performed in the static analysis stage;
- Task execution time: provides the execution time of a task on a specific processing element, according to the mapping solution defined by the previous pragma. Worst-case time can be provided in case the execution time is data-dependent.
- Task reconfiguration time: specifies the reconfiguration time. It is "zero" for SW and static HW implementations. In particular, the processing elements are distinguished into three types: software processing elements (SPE), static-hardware processing elements (HPE), and reconfigurable processing elements (RPE). Reconfiguration time of SPE and HPE is zero, while for the RPE, bitstream loading induces a reconfiguration overhead.

The above information is provided by Task 2.4 of the FASTER tool flow; this is the Task in the baseline scheduler is determined by updating the corresponding part of the XML description.

The aforementioned information describes the system and will be provided to the runtime scheduler in the form of data structures. In particular, five (5) generic data structures are defined to represent the structure and the status of the system following the work presented in (Gianluca Durelli, Christian Pilato, Andrea Cazzaniga, Donatella Sciuto, Marco D. Santambrogio, July 2012). These are: the processing element, the application, the task graph, the task and the implementations.

- Processing element: encodes the information about the different processing elements in the system. For each of them it records:
  - the type: whether an element is a reconfigurable element or not;
  - the status: if a single PE is available, under reconfiguration or which task is executing;
  - the current configuration: current configured task, in case the PE is a reconfigurable element.

  The processing element data structure is generated by analyzing the information included in the input XML.
- Application: encodes the information about each of the applications by including the data about:
  - the status: whether it is running or not;
  - the tasks: a pointer to the tasks that compose the application;
  - the task graph: a pointer to the structure encoding the task graph of the application.
- Task graph: it is represented as an incidence matrix encoding the dependencies between the tasks of the application. This is formed statically and does not change at run time. Moreover, a vector containing the number of dependencies a task must solve before it reaches the ready state, is used. In particular, execution of a task can start once the tasks preceding this task have completed their execution. The values within this vector are updated at run-time. This vector has been introduced in order to perform an action only in case there is an actual update, because performing an analysis of the incidence matrix at every scheduling step is computationally intensive.

- Tasks: contains the information about the tasks of an application, and in particular:
    - the status: if the task is ready, running, to be executed, to be configured or already completed;
    - the implementations: the list of implementations available for the task, their timing estimation (execution and reconfiguration time) and the function to be invoked once the implementation is executed.
- Implementation: contains the information about the implementation, as, for example, the partial bitstream needed in case of a reconfigurable hardware implementation, and the number of times it has been used and reconfigured.

2.5. Statistics to drive scheduler decisions

We intend to use online statistics for driving scheduler's decisions. The following list is indicative but feasible given the capabilities of the platforms studied within the context of the FASTER project:

- A history of the number of times each task has been called will be maintained. Each task will be graded accordingly so as the scheduler can decide on the fly whether a task should be replaced by another in a certain reconfigurable area. Tasks that are called more frequently than others will have higher priority so as to refrain from unloading or replacing them. Caching and prefetching actions can be driven by these statistics; however the initial directions for caching and prefetching can be provided in the XML file as a result of the static analysis.
- A metric that will allow for the runtime system to be aware of the fragmentation level is also considered. In case a scheduling puts the system in a fragmented state, this scheduling should be avoided in the future. Thus keeping statistics of scheduling decisions previously taken that caused fragmentation will prevent the system from becoming fragmented too soon again.
- Temperature is a factor that can be affected by the scheduling decisions. Decisions that cause non-uniform distribution of the temperature and the presence of hot spots would cause shutting-off the chip's operation. To prevent such cases, the values of the temperatures should be observed and collected at runtime. The amount of memory needed to do so should be studied. This will be done in the next phase of WP4.
- Statistics can be also maintained for the execution time of tasks that are data-dependent. To the same direction, task deadlines could be updated by monitoring the performance of each task.
- In order to speed up the configuration process for dynamic circuit specialization, the run-time scheduler needs to be aware of where the template configuration is loaded in the FPGA. This information, i.e. the knowledge of which bitstream is loaded in which area, can be considered as a statistic needed by the process of micro-reconfiguration. In dynamic circuit specialization, reconfiguration is performed in two phases. In the first phase a template configuration is loaded in one of the available reconfiguration areas. In the second phase a partial bitstream is loaded (micro reconfiguration) that specializes the template. If on a later moment another specialized version of the task needs to be executed and the template is still available on the FPGA, the reconfiguration process can be faster since only the micro reconfiguration step needs to be performed.

2.6. Storage requirements

In order for the above information to be accessible by the runtime scheduler, it should be stored in memory. The type of storage resources to be used depends on whether the system targets the desktop or the embedded domain. In the desktop domain the SDRAM memory of the host computer will be used to hold the data structures, while in the embedded domain the SDRAM memory housed on the FPGA platform will be used. For non-volatile memory means, the HDD of the host and the compact flash of the FPGA platform will be used. The ZBT SRAM located in the FPGA platform might be used for caching purposes in both domains.

Small amount of information will be kept internally in the FPGA memories (BRAMs), while status registers will be used for direct access. For example, the status of the PE (e.g., whether a partially reconfigurable area is empty, being configured, ready for execution, or busy executing) will be accessible in status registers. The memory locations and registers holding dynamic data will be updated at run-time. The Task Configuration Microcode (TCM) will be stored internally in the FPGA memories in an embedded system, while in the desktop domain the memory of the host will be used.

Finally, two more cases might need to be covered that require special memory management: first, the case of data exchange amongst tasks that do not co-exist at the same time in the chip, in order for the current task to have access to the results of a previous task not currently configured in the FPGA. The second case is preemptive scheduling in which the context of the task should be saved so as to continue from the point it stopped executing.

With regard to the configuration bitstreams, these will be held in the host memory (for the desktop domain) or the DRAM (for the embedded domain). For non-volatile storage, we will use the HDD of the host and the compact flash of the FPGA platform respectively. With regard to micro-reconfiguration, a micro-reconfigurable task has a parameterized configuration that needs to be stored somewhere in the same place with the regular partial bitstreams. This parameterized configuration can be split up in a template configuration, which is independent of the parameter value, and a specialization function that generates the dependent tasks of the configuration given the parameter values. The parameterized configuration will be stored next to the other bitstreams.

2.7. Level of transparency to the user

The functionality of the run-time scheduler operation will remain completely transparent to the user. Only the basic directions in the XML, i.e. baseline scheduler, should be created by the designer. Information such as number of implementations per task, platform blocks in which a task can be executed (CPU, static HW, reconfigurable part) and so on, are provided at design time in the XML. The tool flow will generate the run-time scheduler that will take into account such considerations.

2.8. Conclusion

We have defined the exact requirements and inputs of the runtime scheduler. The information is provided in the form of data structures. The initial directions for the runtime scheduler will be available in the XML file. This will be saved in different type of storage means, such as the host memory, status registers, and internal memory implemented with BRAMs.

3. Basic Functionality of the Scheduler

This Section discusses the experimental system(s) we have developed which are up and running to demonstrate a basic functionality. Also we discuss possible extensions; some of them will be realized in a following phase of the project.

To study the functionality of the runtime scheduler we have built an FPGA-based system equipped with a Virtex-5 card plugged on the PCIe slot of a PC running a CentOS linux operating system.

This environment is close to that of Maxeler's desktop computer; however it is much simpler and much less expensive. A software application running above the host OS awaits input from the user in order to make a selection through a command line interface. For demonstration purposes we have designed three simple kernels as IP cores for execution in the FPGA; the corresponding bitstreams are stored in the HDD of the host PC. Once the user enters a choice, a user level program triggers reconfiguration of the FPGA by loading the corresponding bitstream; if the choice matches the kernel already loaded into the FPGA, reconfiguration is not triggered. The reconfiguration process remains completely transparent to the user entering the kernel choice. The user receives only the result of kernel execution. In the present system data transactions are performed through DMA achieving a throughput of 1.5 Gbps, which is close to the theoretical bandwidth of PCIe (2Gbps for PCIe lane x1), while reconfiguration is conducted through JTAG interface. This is slow and we will move to faster reconfiguration interfaces (ICAP or SelectMAP).

Also, an embedded system based on the same Virtex-5 platform has been developed in which reconfiguration is controlled by the embedded MicroBlaze through ICAP. Two partially reconfigurable areas have been designated, used for loading/unloading modules per user selection; again this process is performed transparently to the user.

In the above systems the FASTER runtime scheduler will manage core execution, reconfiguration calls and the interrupt routines associated with each core. An interrupt controller will be responsible for detecting the task completions; all cores have an output signal that acts as interrupt to the interrupt controller running in software. Currently, in the desktop system, a simple scheduler loads the bitstream as needed, manages the core execution by sending the input data to it and returns the result to the user.

3.1. Scheduler functionality and algorithms

In the Linux environment, we have developed software that demonstrates the basic functionality of the scheduler. In particular, we have implemented the dining philosopher's problem which resembles a real scenario with tasks (a philosopher represents a task) that should be served. The tasks will be loaded through reconfiguration, waiting to be executed, execute, and unloaded. To address this problem the scheduler treats the tasks as a whole and also each task individually, in order to schedule them, execute them and terminate them. We address the problem with the following set of functions:

- Synchronization functions to ensure safe operation (mutexes, condition variables)
- Function for scheduling (currently we have implemented only the round robin scheduling algorithm; in the future we will implement other algorithms)
- System calls
- Initialization functions (to create the task table, the scheduler, and the initial context)

We created a table in which we store the tasks. Each task has a unique context, which contains the pointer to the stack that holds the data, the size of the task, the contents of the registers and several flags. Presently, the system is demonstrated using the simple round robin scheduling. In case the scheduler gets a notification to alter a task being executed, it stops the current task, it saves the context, it loads the context of the new task, and then it starts its execution. Next, we will evaluate the functionality of the system using other scheduling policies such as the First Come First Served, the Earliest Deadline First, First Fit and possibly a more sophisticated scheduling policy that will take into account the bin-packing problem (Ali Ahmadinia, Christophe Bobda, Jurgen Teich, 2004).

3.2. Communication with the kernels

The kernels are composed of two main parts: the logic and the interface for the communication with the rest of the system. In case the embedded processor is used, the kernels will be connected on the PLB bus. Also, they can be connected to the external memory using the NPI protocol. For the desktop domain we are using the PCIe interface. The reconfigurable core is controlled by a FSM, which in turn receives/sends data from/to the PCIe using FIFOs. This structure is used in the desktop system described above and has been presented in detail in (Kyprianos Papadimitriou, Charalampos Vatsolakis, Dionisios Pnevmatikatos, July 2012). The host communicates with the FPGA with DMA transactions. The FPGA device is the bus master; a read DMA transaction performs the data copy from the main memory to the FPGA, whereas a write DMA transaction copies data from the FPGA to the main memory. In particular, in order to send data to the device via DMA, the data are copied to the memory and a DMA read is performed. In order to receive data from the FPGA device, we have to perform a DMA write and copy the data to the user space. With regard to the reconfiguration of the kernels, this is conducted through the JTAG port using a Xilinx programmer connected to the USB port of the Host PC.

3.3. Scheduling decisions based on application state

The scheduler should also react accordingly to the application state. Next, we focus on certain cases in which we specify the scheduler reactions or options:

- When the execution of a task placed in a reconfigurable area is completed, in case there is no scheduled decision for this reconfigurable area there are three options: either leaving undisturbed the area with the task idle, or, reconfiguring this area with another task, or erasing the reconfigurable area;
- If the host application requires access to an IP already placed in a partially reconfigurable area, the runtime scheduler should be aware of this information in order to avoid scheduling a reconfiguration. This information will be held in a status register accessible by the scheduler. Our desktop system described above already supports this functionality. Depending on the kernel loaded in the reconfigurable fabric, a status register is updated (e.g., for the three kernels we decoded this as 00 (when nothing is loaded), 01 (addition kernel), 10 (subtraction kernel), and 11 (LED toggling kernel));
- Prefetching and caching operations will be executed depending on the application's state to satisfy temporal locality. Also, the same operations are considered for covering cases in which a hardware task is likely to be required in the future, e.g. branch that has been selected and it is very likely that a hardware task lying under this branch will be executed in the near future. A decision will be made whether the bitstream that corresponds to the IP will be cached in certain level of the memory hierarchy such as the external DRAM, SRAM, on BRAM of the FPGA, or even prefetched in the FPGA configuration memory.

3.4. Scheduling decisions driven by non functional data

The decisions of the scheduler will be also driven by non-functional data. Three are the main inputs that will affect the decisions of the scheduler: power consumption, temperature and the current arrangement of tasks in the FPGA area. These data will be accessible to the runtime scheduler through data structures.

3.5. Micro-reconfiguration support

A Dynamic Circuit Specialization (DCS) task is performed in two phases. In the first phase, the parameter input data is used to generate a specialized configuration. In the second phase, this specialized task is executed on the FPGA. These two phases can be seen as two subtasks: the specialization task and the computation task. The specialization task is most likely a software task and is dependent on the availability of the parameter data. The computation task is an FPGA task and depends on the regular input data and the availability of the specialized configuration generated by the specialization task.

As was mentioned before, reconfiguration of a DCS task is also done in two phases: loading the template and loading the specialized configuration. If a DCS task is invoked several times the reconfiguration time can be drastically reduced if the template stays available. The runtime system should thus have a mechanism that preserves the template when additional executions of the DCS task are expected.

Care should be taken in the way the control of parameter changing will be done. The RTSM should include software able to determine the current status of the system, maybe by using lookup tables. We need to specify the data structures that will be needed to do so. Finally, the access to configuration port will be controlled directly by the RTSM and not by the software that controls the parameter changing.

3.6. Integration with the run-time system of the Maxeler platform

In the Maxeler system, the runtime system manager called "accelerator management software" is available as an RPM software package. Kernel device driver, status monitoring daemon and runtime libraries are all included in this package. MaxelerOS management software coordinates resource use, scheduling and data movement within the dataflow compute environment. Multiple applications can make use of the dataflow compute engines within a system: the system management software (MaxelerOS running within standard Linux) manages the available dataflow compute engines, allocating engines to waiting processes as they are freed-up by other applications and time-slicing resources in the same way a conventional operating system would for conventional CPUs. Reconfiguration of the FPGA chips is managed automatically as necessary to minimize overhead. The MaxCompiler compiles kernels and the manager into a single HW accelerator file, the .max file, which is then linked with the CPU code objects to create a single executable file. The .max file contains the FPGA bistream configuration as well as meta-data to allow it to be used at runtime. When the executable is run, any FPGAs utilized by the application are automatically configured with the correct bitstream. The FASTER tool chain will communicate with the MaxCompiler through a specific interface in order for the FASTER runtime scheduler to intervene in the process (Deliverable_1.2, May 2012).

4. Implementation of the Scheduler

The scheduler will be implemented in Linux. Whether it will be implemented as an OS extension or as a standalone library out of the kernel is a decision trading off flexibility and performance.

The scheduler is governed by the runtime system manager, which needs to be generated every time a new application or a new set of applications are developed with the FASTER framework. Thus, it is reasonable to create the scheduler outside of the Linux kernel as a standard library. The runtime system manager of Maxeler is implemented as an RPM library, so we need to study the way it will communicate with the FASTER RTSM.

The runtime system is required to execute the compiled code. The compiler can communicate with the RTSM through procedures and function calls, without direct reference to the RTSM data structures aside from the parameters of the RTSM subprograms. The RTSM data structures may be kept in a separate address space, protected from access by the application. The direction of call is always from application code to the RTSM. However, there are some exceptions: Task creation in which the compiler passes to the RTSM the address of a procedure corresponding to the task, and, protected entries in which the compiler passes to the RTSM the address of an array with the reference to the subprograms generated by the compiler.

Figure 3 illustrates the Runtime hierarchy. The program requests the services of the runtime system through subprograms calls. The API provides the interface to request services from the Operating System.

Program
Runtime library
API
Operating System
Hardware

Figure 3 Runtime Hierarchy

5. References

Ali Ahmadinia, Christophe Bobda, Jurgen Teich. (2004). A Dynamic Scheduling and Placement Algorithm for Reconfigurable Hardware. *International Conference of Architecture on Computing Systems (ARCS)*, (pp. 125-139).

Deliverable_1.2. (May 2012). *Requirements of FASTER Models, Methods and Tools.* FASTER Project.

Deliverable_2.2. (Aug 2012). *Initial Partitioning Methodology for Reconfigurable Systems.* FASTER Project.

Deliverable_4.1. (Feb 2012). *Evaluation of Existing Run-Time System Support for Reconfiguration.* FASTER Project.

Deliverable_4.2. (Aug 2012). *Architectural Extensions for Run-Time System Support.* FASTER Project.

Gianluca Durelli, Christian Pilato, Andrea Cazzaniga, Donatella Sciuto, Marco D. Santambrogio. (July 2012). Automatic Run-Time Manager Generation for Reconfigurable MPSoC Architecture. *International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC).* York.

Kyprianos Papadimitriou, Charalampos Vatsolakis, Dionisios Pnevmatikatos. (July 2012). Acceleration of Computationally-Intensive Kernels in the Reconfigurable Era. *International Workshop on Reconfigurable and Communication-centric Systems-on-Chip (ReCoSoC).* York.

Marconi, T. (2011). *Efficient Runtime Management of Reconfigurable Hardware Resources, Ph.D Thesis.* TUDelft.