Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2009, Article ID 976296, 13 pages
doi:10.1155/2009/976296

Research Article

Run-Time HW/SW Scheduling of Data Flow Applications on Reconfigurable Architectures

Fakhreddine Ghaffari, Benoit Miramond, and François Verdier

ETIS Laboratory, UMR 8051, ENSEA, University of Cergy Pontoise, CNRS, 6 avenue Du Ponceau, BP 44, 95014 Cergy-Pontoise Cedex, France

Correspondence should be addressed to Fakhreddine Ghaffari, fakhreddine.ghaffari@ensea.fr

Received 1 March 2009; Revised 22 July 2009; Accepted 7 October 2009

Recommended by Markus Rupp

This paper presents an efficient dynamic and run-time Hardware/Software scheduling approach. This scheduling heuristic consists in mapping online the different tasks of a highly dynamic application in such a way that the total execution time is minimized. We consider soft real-time data flow graph oriented applications for which the execution time is function of the input data nature. The target architecture is composed of two processors connected to a dynamically reconfigurable hardware accelerator. Our approach takes advantage of the reconfiguration property of the considered architecture to adapt the treatment to the system dynamics. We compare our heuristic with another similar approach. We present the results of our scheduling method on several image processing applications. Our experiments include simulation and synthesis results on a Virtex V-based platform. These results show a better performance against existing methods.

1. Introduction

One of the main steps of the HW/SW codesign of a mixed electronic system (Software and Hardware) is the scheduling of the application tasks on the processing elements (PEs) of the platform. The scheduling of an application formed by N tasks on M target processing units consists in finding the realizable partitioning in which the N tasks are launched onto their corresponding M units and an ordering on each PE for which the total execution time of the application meets the real-time constraints. This problem of multiprocessor scheduling is known to be NP-hard [1, 2], that is, why we propose a heuristic approach.

Many applications, in particular in image processing (e.g., an intelligent embedded camera), have dependent data execution times according to the nature of the input to be processed. In this kind of application, the implementation is often stressed by real-time constraints, which demand adaptive computation capabilities. In this case, according to the nature of the input data, the system must adapt its behaviour to the dynamics of the evolution of the data and continue to meet the variable needs of required calculation (in quantity and/or in type). Examples of applications where the processing needs changes in quantity (the computation load is variable) come from the intelligent image processing where the duration of the treatments can depend on the number of the objects in the image (motion detection, tracking, etc.) or of the number of interest areas (contours detection, labelling, etc.).

We can quote also the use of run time of different filters according to the texture of the processed image (here it is the type of processing which is variable). Another example of the dynamic applications is video encoding where the run-length encoding (RLE) of frames depends on the information within frames.

For these dynamic applications, many implementation ways are possible. In this paper we consider an intelligent embedded camera for which we propose a new design approach compared to classical worst case implementations.

Our method consists in evaluating online the application context and adapting its implementation onto the different targeted processing units by launching a run time partitioning algorithm. The online modification of the partitioning result can also be a solution of fault tolerance, by affecting in run time the tasks of the fault target unit on others operational targets [3]. This induces also to revise the scheduling strategy. More precisely, the result of this later must change at run time in two cases.

(i) Firstly, to evaluate the partitioning result. After each modification of the tasks implementations we need to know the new total execution time. And this is only possible by rescheduling all the tasks.

(ii) Secondly, by modifying the scheduling result we can obtain a better total execution time which meets the real-time constraint without modifying the partitioning. This is because the characteristics of the tasks (mainly execution time) are modified according to the nature of the input data.

In that context, the choice of the implementation of the scheduler is of major importance and depends on the heuristic complexity. Indeed, with our method the decisions taken online by our scheduler can be very time consuming. A software implementation of the proposed scheduling strategies will then delay the application tasks. For this reason, we propose in this work a hardware implementation for our scheduling heuristic.

With this implementation, the scheduler takes only few clock cycles. So we can easily call the scheduler at run time without penalty on the total execution time of the application.

The primary contribution of our work is the concept of an efficient online scheduling heuristic for heterogeneous multiprocessor platforms. This heuristic provides good results for both hardware tasks (onto the FPGA) and software tasks (onto the targeted General Purpose Processors) as well as an extensive speedup through the hardware implementation of this scheduling heuristic. Finally, the implementation of our scheduler allows the system to adapt itself to the application context in real time. We have simulated and synthesized our scheduler by targeting a FPGA (Xilinx Virtex 5) platform. We have tested the scheduling technique on several image processing applications implemented onto a heterogeneous target architecture composed of two processors coupled with a configurable logic unit (FPGA).

The remainder of this paper is organized as follows. Section 2 presents related works on hardware/software scheduling approaches. Section 3 introduces the framework of our scheduling problem. Section 4 presents the proposed approach. Section 5 shows the experimental results, and finally Section 6 concludes this paper.

2. Related Works

The field of study which tries to find an execution order for a set of tasks that meets system design objectives (e.g., minimize the total application execution time) has been widely covered in the literature. In [4–6] the problem of HW/SW scheduling for system-on-chip platforms with dynamically reconfigurable logic architecture is exhaustively studied. Moreover several works deal with scheduling algorithm implemented in hardware [7–9]. Scheduling in such systems is based on priorities. Therefore, an obvious solution is to implement priorities queues. Many hardware architectures for the queues have been proposed: binary tree comparators, FIFO queues plus a priority encoder, and a systolic array priority queue [7]. Nevertheless, all these approaches are based on a fixed priority static scheduling technique. Moreover most of the hardware proposed approaches addresses the implementation of only one scheduling algorithm (e.g., Earliest Deadline First) [9, 10]. Hence they are inefficient and not appropriate for systems where the required scheduling behavior changes during run time. Also, system performance for tasks with data dependent execution times should be improved by using dynamic schedulers instead of static (at compile time) scheduling techniques [11, 12].

In our work, we propose a new hardware implemented approach which computes at run-time tasks priorities based on the characteristics of each task (execution time, graph dependencies, etc.). Our approach is dynamic in the sense that the execution order is decided at run time and supports a heterogeneous (HW/SW) multiprocessor architecture.

The idea of dynamic partitioning/scheduling is based on the dynamic reconfiguration of the target architecture. Increasingly FPGA [13, 14] offers very attractive reconfiguration capabilities: partial or total, static or dynamic.

The reconfiguration latency of dynamically reconfigurable devices represents a major problem that must not be neglected. Several references can be found addressing temporal partitioning for reconfiguration latency minimization [15]. Moreover, configuration prefetching techniques are used to minimize reconfiguration overhead. A similar technique to lighten this overhead is developed in [16] and is integrated into an existing design environment. A prefetch and replacement unit modifies the schedule and significantly reduces the latency even for highly dynamic tasks.

In fact, there are two different approaches in the literature: the first approach reduces reconfiguration overhead by modifying scheduling results.

The second one distinguishes between scheduling and reconfiguration. The reconfiguration occurs only if the HW/SW partitioning step needs it. The scheduling algorithm is needed only to validate this partitioning result.

After partitioning, the implementation of each task is unchanged and configuration is not longer necessary. Scheduling aims at finding the best execution time for a given implementation strategy. Since scheduling does not change partitioning decision it does not take reconfiguration time into account.

In this paper, we focus only on the scheduling strategy in the second case. We assume that the reconfiguration aspects are taken into account during the HW/SW partitioning step (decision of task implementation). Furthermore we addressed this last step in our previous works [17].

3. Problem Definition

3.1. Target Architecture. The target architecture is depicted in Figure 1. It is a heterogeneous architecture, which contains two software processing units: a Master Processor and a Slave

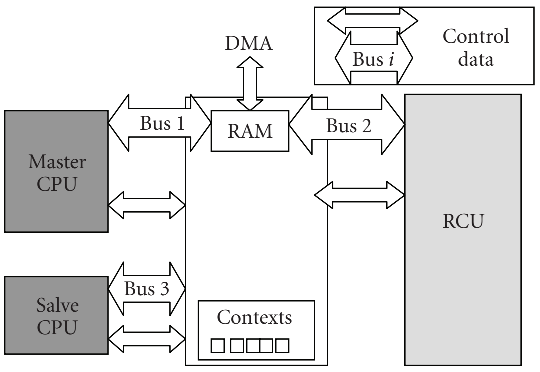


Figure 1: The target architecture.

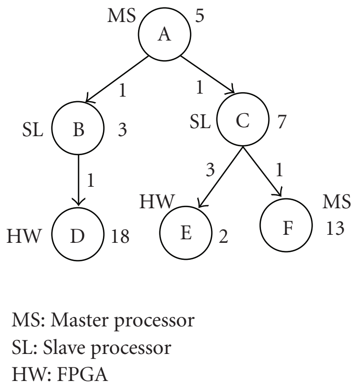


Figure 2: An Example of DFG with 6 tasks.

Processor. The platform also contains a hardware processing unit: (Reconfigurable Computing Unit) RCU and shared memory resources. The software processing units are Von-Neumann monoprocessing systems and execute only a single task at a time.

Each hardware task (implemented on the RCU) occupies a tile on the reconfigurable area [18]. The size of the tile is the same for all the tasks to facilitate the placement and routing of the RCU. We choose, for example, the tile size of the task which uses the maximum of resources on the RCU (we designate by "resource" here the Logic Element used by the RCU to map any task).

The RCU unit can be reconfigured partially or totally. Each hardware task is represented by a partial bitstream. All bitstreams are memorized in the contexts memory (the shared memory between the processors and the RCU in Figure 1). These bitstreams will be loaded in the RCU before scheduling to reconfigure the FPGA according to run-time partitioning results [17]. The HW/SW partitioning result can change at run time according to temporal characteristics of tasks [6]. In [17] we proposed an HW/SW partitioning approach based on HW → SW and SW → HW tasks migrations. The theory of tasks migrations consists in accelerating the task(s) which become critical by modifying their implementations from software units to hardware units and to decelerate the tasks which become noncritical by returning them to the software units.

After each new HW/SW partitioning result, the scheduler must provide an evaluation for this solution by providing the corresponding total execution time. Thus it presents a real-time constraint since it will be launched at run time. With this approach of dynamic partitioning/scheduling the target architecture will be very flexible. It can self-adapt even with very dynamic applications.

3.2. Application Model. The considered applications are data flow oriented applications such as image processing, audio processing, or video processing. To model this kind of applications we consider a Data Flow Graph (DFG) (an example is depicted in Figure 2) which is a directed acyclic graph where nodes are processing functions and edges describe communication between tasks (data dependencies between tasks). The size of the DFG depends on the functional partitioning of the application and then on the number of tasks and edges. We can notice that the structure of the DFG has a great effect on the execution time of the scheduling operations. A low granularity DFG makes the system easy to be predictable because tasks execution time does not vary considerably, thus limiting timing constraints violation. On the other hand, for a very low granularity DFG, the number of tasks in a DFG of great size explodes, and the communications between tasks become unmanageable.

Each node of the DFG represents a specific task in the application. For each task there can be up to three different implementations: Hardware implementations (HW) placed in the FPGA, Software implementations running on the master processor (MS), and another Software implementation running on the slave processor (SL).

Each node of Figure 2 is annotated with two data: one about the implementation (MS or SL or HW) and the other is the execution time of the task. Similarly each edge is annotated with the communication time between two nodes (two tasks).

Each task of the DFG is characterized by the following four parameters:

(a) Texe (execution time),

(b) Impl (implementation on the RCU or on the master processor or on the slave processor),

(c) Nbpred (number of predecessor tasks),

(d) The Nbsucc (number of successor tasks).

All the tasks of a DFG are thus modeled identically, and the only real-time constraint is on the total execution time. At each scheduler invocation, this total execution time corresponds to the longest path in the mapped task graph. It then depends both on the application partitioning and on the chosen order of execution on processors.

4. Proposed Approach

The applications are periodic. In one period, all the tasks of the DFG must be executed. In the image processing, for instance, the period is the execution time needed to

```
For all Software tasks do
{
    Comput ASAP
            Task with minimum ASAP will be chosen
    If (Equality of ASAP)
    Compute Urgency
            Task with maximum urgency will be chosen
    If (Equality of Urgency)
    Compare Execution time
            Task with maximum execution time will be chosen
}
```

Algorithm 1: Principle of our scheduling policy.

process one image. The scheduling must occur online at the end of the execution of all the tasks, and when a violation of real-time constraints is predicted. Hence the result of partitioning/scheduling will be applied on the next period (next image, for image processing applications).

Our run-time scheduling policy is dynamic since the execution order of application tasks is decided at run time. For the tasks implemented on the RCU, we assume that the hardware resources are sufficient to execute in parallel all hardware tasks chosen by the partitioning step. Therefore the only condition for launching their execution is the satisfaction of all data dependencies. That is to say, a task may begin execution only after all its incoming edges have been executed.

For the tasks implemented on the software processors, the conditions for launching are the following.

(1) The satisfaction of all data dependencies.

(2) The discharge of the software unit.

Hereby the task can have four different states.

(i) Waiting.

(ii) Running.

(iii) Ready.

(iv) Stopped.

The task is in the waiting state when it waits the end of execution of one or several predecessor tasks. When a software processing unit has finished the execution of a task, new tasks may become ready for execution if all their dependencies have been completed of course.

The task can be stopped in the case of preemption or after finishing its execution.

The states of the processing units (SW, SL, and HW) in our target architecture are: execution state, reconfiguration state or idle state.

In the following, we will explain the principle of our approach as well as a hardware implementation of the proposed HW/SW scheduler.

As explained in Algorithm 1, the basic idea of our heuristic of scheduling is to take decision of tasks priorities according to three criteria.

The first criterion is the As Soon As Possible (ASAP) time. The task which has the shortest ASAP date will be launched first.

The second criterion is the urgency time: the task which has the maximum of urgency will have priority to be launched before the others. This new criterion is based on the nature of the successors of the task. The urgency criterion is employed only if there is equality of the first criterion for at least two tasks. If there is still equality of this second criterion we compare the last criterion which is execution time of the tasks. We choose the task which has the upper execution time to launch first.

We use these criteria to choose between two or several software tasks (on the Master or on the Slave) for running.

4.0.1. The Urgency Criterion. The urgency criterion is based on the implementation of tasks and the implementations of their successors. A task is considered as urgent when it is implemented on the software unit (Master or slave) and has one or more successor tasks implemented on other different units (hardware unit or software unit).

Figure 3 shows three examples of DFG. In Figure 3(a) task C is implemented on the Slave processor and it is followed by task D which is implemented on the RCU. Thus the urgency (Urg) of task C is the execution time of its successor (Urg (C) = 13). In example (b) it is the task B which is followed by the task D implemented on a different unit (on the Master processor). In the last example (c) both tasks B and C are urgent but task B is more urgent than task C since its successor has an execution time upper than the execution time of the successor of task C.

When a task has several successors with different implementations, the urgency is the maximum of execution times of the successors.

In general case, when the direct successor of task A has the same implementation as A and has a successor with a different implementation, then this last feedbacks the urgency to task A.

We show the scheduling result for case (a) when we respect the urgency criterion in Figure 3(d) and otherwise in Figure 3(e). We can notice for all the examples of DFG in Figure 3 that the urgency criterion makes a best choice to obtain a minimum total execution time. The third criterion (the execution time) is an arbitrary choice and has very rarely impact on the total execution time.

We can notice also that our scheduler supports the dynamic creation and deletion of tasks. These online services are only possible when keeping a fixed structure of the DFG along the execution. In that case the dependencies between tasks are known a priori. Dynamic deletion is then possible by assigning a null execution time to the tasks which are not active. and dynamic creation by assigning their execution time when they become active.

This scheduling strategy needs an online computation of several criterions for all software tasks in the DFG.

We tried first to implement this new scheduling policy on a processor. Figure 4 shows the computation time of our scheduling method when implemented on an Intel Core 2

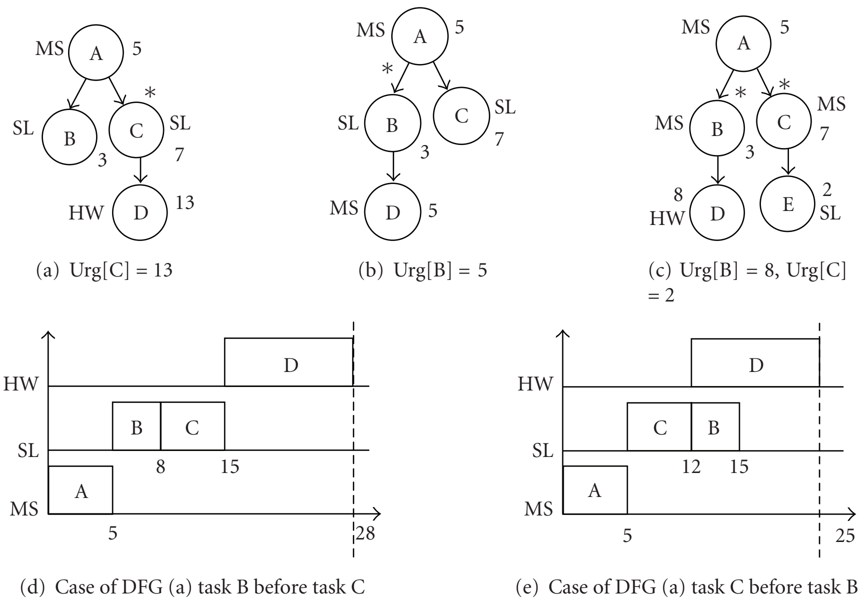

MS
A
5
SL
B
3
C
SL
7
HW
D
13
(a) Urg[C] = 13
(b) Urg[B] = 5
(c) Urg[B] = 8, Urg[C] = 2
HW
SL
MS
D
B
C
A
8
15
5
28
12
25
(d) Case of DFG (a) task B before task C
(e) Case of DFG (a) task C before task B

Figure 3: Case examples of urgency computing.

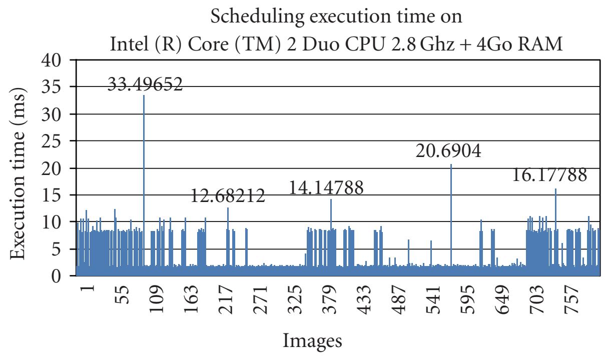


Figure 4: Execution time of the software implementation of the scheduler.

Duo CPU with a frequency of 2.8 GHz and 4 Go of RAM. We can notice that the average computation time of the scheduler is about 12 milliseconds for an image. These experiments are done on an image processing application (the DFG depicted on Figure 12) whose period of processing by an image is 19 milliseconds. So the scheduling (with this software implementation) takes about 63% of a one image processing computation time on a desktop computer.

We can conclude that, in an embedded context, a software implementation of this strategy is thus incompatible with real-time constraints.

We describe in the following an optimized hardware implementation of our scheduler.

4.1. Hardware Scheduler Architecture. In this section, we describe the proposed architecture of our scheduler. This architecture is shown in Figure 5 for a DFG example of three tasks. It is divided in four main parts.

(1) The DFG_IP_Sched (the middle part surrounded by a dashed line in the figure).

(2) The DFG_Update (DFG_Up in the figure).

(3) The MS_Manager (SWTM).

(4) The Slave_Manager (SLTM).

The basic idea of this hardware architecture is to parallelize at the maximum the scheduling of processing tasks. So, at the most (and in the best case), we can schedule all the tasks of the DFG in parallel for infinite resources architecture.

We associate to the application DFG a modified graph with the same structure composed of the IP nodes (each IP represents a task). Therefore in the best case, where tasks are independent, we could schedule all the tasks in the DFG in only one clock cycle.

To parallelize also the management of the software execution times, we associate for each software unit a hardware module:

(i) the Master Task Manager (SWTM in Figure 5),

(ii) the Slave Task Manager (SLTM in the Figure 5).

These two modules manage the order of the tasks executions and compute the processor execution time for each one.

The inputs signals of this scheduler architecture are the following.

(i) A pointer in memory to the implementations of all the tasks. We have three kinds of implementation (RCU, Master, and Slave). With the signals SW and HW we can code these three possibilities.

(ii) The measured execution time of each task (Texe).

(iii) The Clock signal and the Reset.

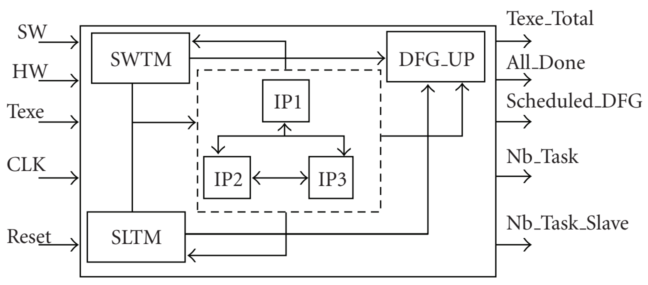


Figure 5: An example of the scheduler architecture for a DFG of three tasks.

The outputs signals are the following.

(i) The total execution time after scheduling all tasks (Texe_Total).

(ii) The signal All_Done which indicates the end of the scheduling.

(iii) Scheduled_DFG is a pointer to the scheduling result matrix to be sent to the operating system (or any simple executive).

(iv) The Nb_Task and the Nb_Task_Slave are the number of tasks scheduled on the Master and the number of tasks scheduled on the Slave, respectively. As noted here, these two signals were added solely for the purpose of simulation in ModelSim (to check the scheduling result). In the real case we do not need these two output signals since this information comes from the partitioning block.

The last one is the DFG_Up. This allows updating the results matrix after each scheduling of a task.

In the following paragraphs, we will detail each part of this architecture.

4.1.1. The DFG_IP_Sched Block. In this block there are N components (N is the number of tasks in the application). For each task we associate an IP component which computes the intrinsic characteristics of this task (urgency, ASAP, Ready state, etc.). It also computes the total execution time for the entire graph.

The proposed architecture of this IP is shown in Figure 6 (in the appendix).

For each task the implementation PE and the execution time are fixed, so the role of this IP is to calculate the start time of the task and to define its state. This is done by taking into account the state of the corresponding target (master, slave, or RCU). It then iterates along the DFG structure to determine a total execution ordering and to affect the start time.

This IP calculate also the urgency criterion of critical tasks according to the implementation and the execution time of their successors.

If the task is implemented on the RCU it will be launched as soon as all its predecessors will be done. So the scheduling time of hardware tasks depends on the number of tasks that we can run in parallel. For example, the IP can schedule all hardware tasks that can run in parallel in a one clock cycle.

For the software tasks (on the master or on the slave) the scheduling will take one clock cycle per task. Thus the computing time of the hardware scheduler only depends on the result of the HW/SW partitioning.

4.1.2. The DFG_Update Block. When a DFG is scheduled the result modifies the DFG into a new structure. The DFG_Update block (Figure 7 in the appendix) generates new edges (dependencies between tasks) after scheduling in objective to give a total order of execution on each computing unit according to the scheduling results.

We represent dependencies between tasks in the DFG by a matrix where the rows represent the successors and the columns represent the predecessors. For example, Figure 8 depicts the matrix of dependencies corresponding to the DFG of Figure 2. After scheduling, the resulting matrix is the update of the original one. It contains more dependencies than this later. This is the role of the DFG_Update block.

4.1.3. The MS_Manager Block. The objective of this module is to schedule the software tasks according to the algorithm given above. Figure 9 in the appendix presents the architecture of the Master Manager bloc. The input signal ASAP_SW represents the ASAP times of all the tasks. The Urgency_Time signal represents the urgency of each task of the application. The SW_Ready signal represents the Ready signals of all the software tasks.

The Signal MIN_ASAP_TASKS represents all the tasks "Ready" and having the same minimum values of time ASAP.

The signal MAX_CT_TASKS represents all the tasks "Ready" and having the same maximum of urgency. The tasks which have the two preceding criteria will be represented by the Tasks_Ready signal. The Task_Scheduled signal determines the only software task which will be scheduled. With this signal, it is possible to choose the good value of signal TEXE_SW and to give the new value of the SW_Total_Time signal thereafter. A single clock cycle is necessary to schedule a single software task.

By analogy the Slave_Manager block has the same role as the SW_Manager block. From scheduling point of view there is no difference between the two processors.

4.2. HW/SW Scheduler Outputs. In this section, we describe how the results of our scheduler are processed by a target module such as an executive or a Real-Time Operating System (RTOS). As depicted in Figure 8 , the output of our run-time HW/SW scheduler is $n \times n$ matrix where "n" is the total number of tasks in the DFG. Figure 10 shows the scheduling result of the DFG depicted in Figure 12. This matrix will be used by a centralized Operating System (OS) to fill its task queues for the three computing units.

The table shown in Figure 11 is a compilation of both the results of the partitioning and scheduling operations.

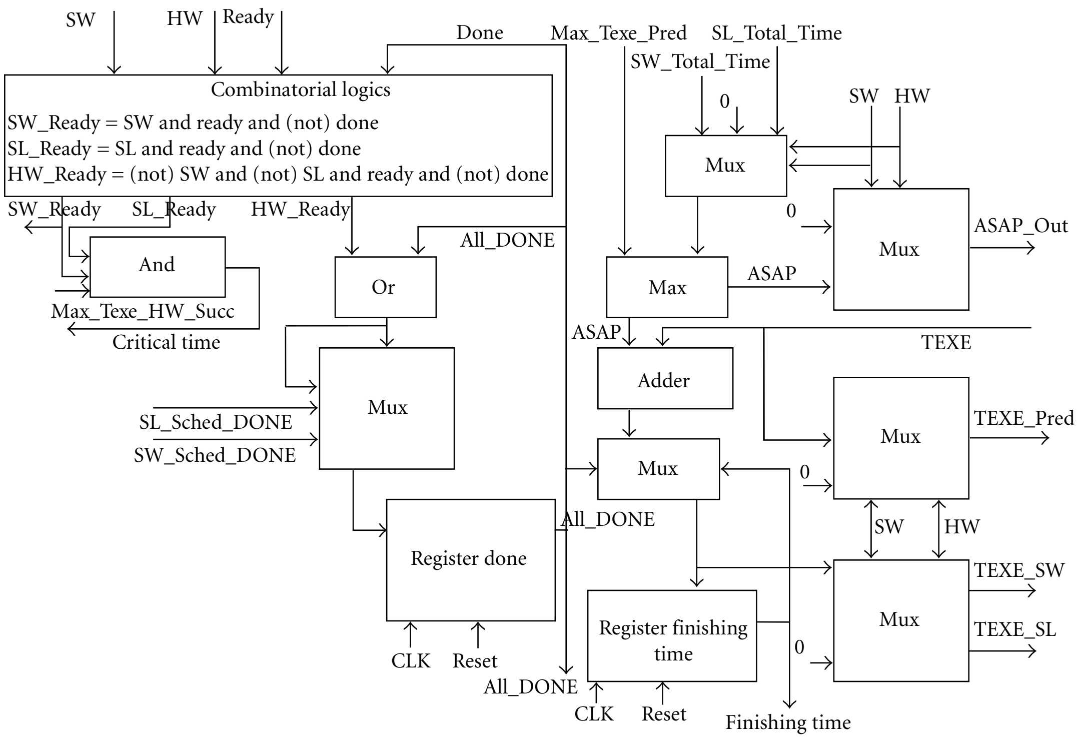


Figure 6: An IP representing one task.

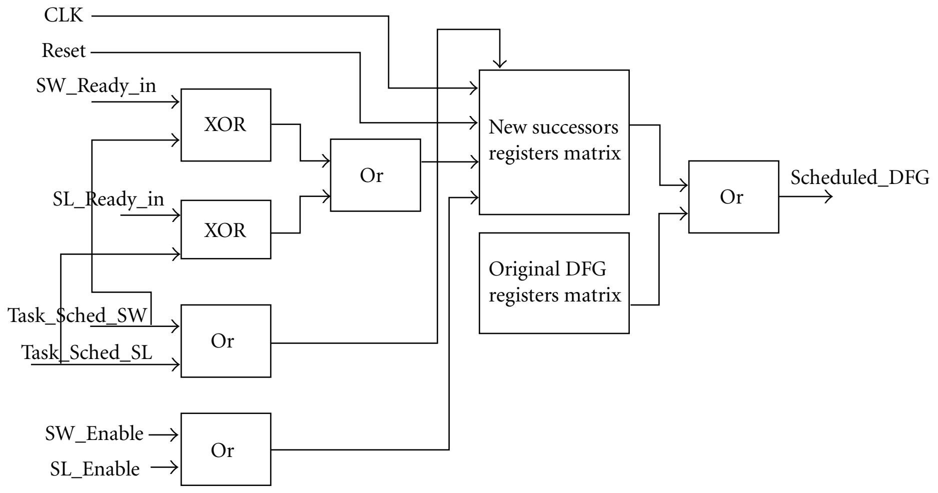


Figure 7: The DFG updating architecture.

The OS browses the matrix row by row. Whenever it finds a "1" it passes the task whose number corresponds to the column in the waiting state. At the end of a task execution the corresponding waiting tasks on each units will become either Ready or Running.

A task will be in the Ready state only when all its dependencies are done and that the target unit is busy. Thus there is no Ready state for the hardware tasks.

It should be noted that if the OS runs on the Master processor, for example, this later will be interrupted each time to execute the OS.

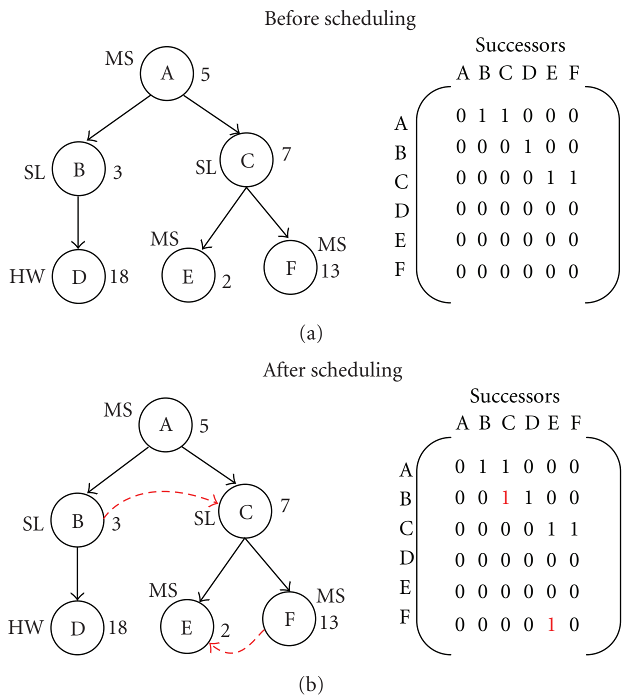
5. Experiments and Results

With the idea to cover a wide range of data-flow applications, we leaded experiments on real and artificial applications. In the context of this paper we present the summary of the results obtained on a 3-case studies in the domain of real-time image processing:

(i) a motion detection application,

(ii) an artificial extension of this detection application,

(iii) a robotic vision application.



$$\begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & 0 & 1 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 0 & 1 & 0 & 0 \\ C & 0 & 0 & 0 & 0 & 1 & 1 \\ D & 0 & 0 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 0 & 0 & 0 \\ F & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$\begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & 0 & 1 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 1 & 0 & 0 \\ C & 0 & 0 & 0 & 0 & 1 & 1 \\ D & 0 & 0 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 0 & 0 & 0 \\ F & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Figure 8: Matrix result after scheduling.

The second case study is a complex DFG which contains different classical structures (Fork, join, sequential). This DFG is depicted in Figure 12. It contains twenty tasks. Each task can be implemented on the Software computation unit (Master or Slave processor) or on the Reconfigurable RCU. The original DFG is the model of an image processing application: motion detection on a fixed image background. This application is composed of 10 sequential tasks (from ID 1 to ID 10 in Figure 12). We added 10 others virtual tasks to obtain a complex DFG containing the different possible parallel structures. This type of parallel program paradigm (Fork, join, etc.) arises in many application areas.

In order to test the presented scheduling approach, we have performed a large number of experiments where several scenarios of HW/SW partitioning results were analyzed.

As an example, Figure 12 presents the scheduling result when tasks 3, 4, 7, 8, 11, 12 17, 18, and 20 are implemented in hardware. As explained in Section 4.1 new dependencies (dotted lines) are added in the original graph to impose a total order on each processor. In this figure all the execution times are in milliseconds (ms).

We also leaded our experiments on a more dynamic application from robotic vision domain [19]. It consists in a subset of a cognitive system allowing a robot equipped with a CCD-camera to navigate and to perceive objects. The global architecture in which the visual system is integrated is biologically inspired and based on the interactions between the processing of the visual flow and the robot movements. In order to learn its environment the system identifies keypoints in the landscape.

Keypoints are detected in a sampled scale space based on an image pyramid as presented in Figure 13. The application is dynamic in the sense that the number of keypoints depends on the scene observed by the camera. Then the execution time of the Search and Extract tasks in the graph dynamically changes (see [19] for more details about this application).

5.1. Comparison Results. Throughout our experiments, we compared the result of our scheduler with the one given by the HCP algorithm (Heterogeneous Critical Path) developed by Bjorn-Jorgensen and Madsen [20]. This algorithm represents an approach of scheduling on a heterogeneous multiprocessor architecture. It starts with the calculation of priorities for each task associated with a processor. A task is chosen depending on the length of its critical path (CPL). The task which has the largest minimum CPL will have the highest priority. We compared with this method because it is shown better than several other approaches (MD, MCP, PC, etc.) [21].

The summary of the experiments leaded is presented in Figure 14. Each column gives average values for one of the three presented applications with different partitioning strategies.

By comparing the first and second rows, our scheduling method provides consistent results.

The quality of the scheduling solutions found by our method and the HCP method is similar. Moreover, our method obtains better results for the complex Icam application. The HCP method returns an average total execution time equal to 69 milliseconds whereas our method returns only 58 milliseconds for the same DFG. For the icam_simple application, the DFG is completely sequential, so whatever the scheduling method the result is always the same. For the robotic vision application, we find the same total execution time with the two methods because of the existence of a critical path in the DFG which always sets the overall execution time. We also measured the execution overhead of the proposed scheduling algorithm when it is implemented in software (third row of Figure 14) and in hardware (forth row).

Since the scheduling overhead depends on the number of tasks in the application we only indicate the average values in Figure 14. For example, Figure 15 presents the execution time of the hardware scheduler (in cycles) according to the number of software tasks.

From Figure 15, it may be concluded that when the result of partitioning changes at runtime, then the computation time needed for our scheduler to schedule all the DFG tasks is widely dependent on this modification of tasks implementations. So:

(1) there is a great impact of the partitioning result and the DFG structure on the scheduler computation time,

(2) the longest sequential sequence of tasks corresponds to the case where all tasks are on the Software (each task takes one clock cycle). This case corresponds to the maximum of schedule computation time,

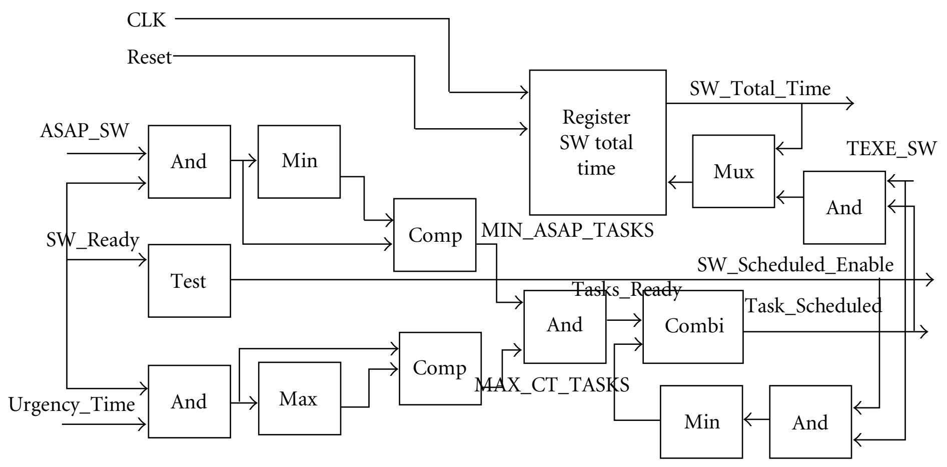


Figure 9: The module of the MS Manager.

Table 1: Device utilization summary after synthesis.

Used logic utilization	Icam_simple	Icam_complex	Robotic_vision
Number of slices registers	<1%	<1%	<1%
Number of slice LUTs	3%	6%	9%
Number of fully used Bit Slices	5%	5%	6%
Number of bonded IOBs	24%	64%	100%
Number of BUFG/BUFGCTRLs	3%	3%	3%
Scheduler frequency	23,94 Mhz	19,54 MHz	17,44 MHz

$$\begin{array}{c|cccccccccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\
\hline
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
12 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
15 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
17 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
18 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
19 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
20 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Figure 10: Scheduling result.

(3) the minimum schedule computation time depends on the DFG structure. The longest sequential sequence of tasks when all tasks are on the hardware (each task takes one clock cycle).

In our case (Figure 12) this sequence is formed by 10 tasks, so the minimum schedule computation time is equal to 10 clock cycles for this application.

The results confirm that a software implementation of our scheduling algorithm is incompatible with online scheduling. Instead, the hardware implementation proposed in the paper brings determinism, a better scalability, and an ×20000 speedup.

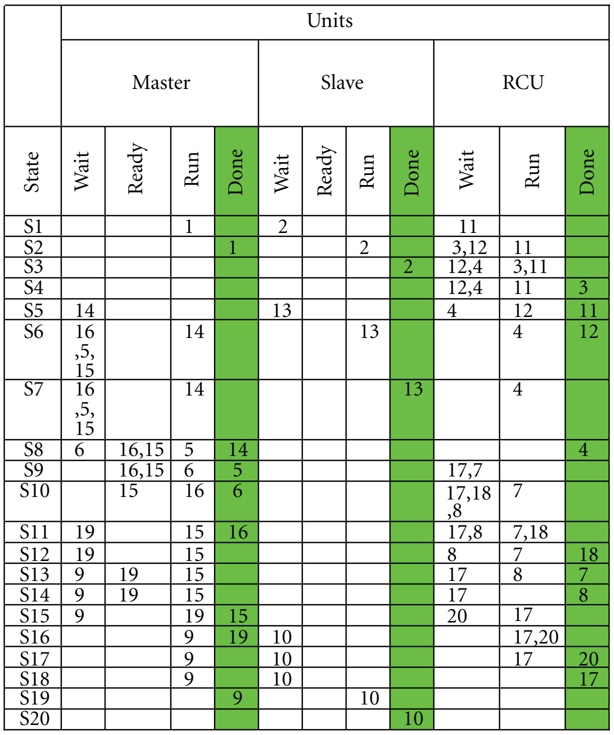
5.2. Synthesis Results. We have synthesized our scheduler architecture with an FPGA target platform (Virtex5, device XC5VLX330 -2 ff 1760) [22] for the RCU of Figure 1. Table 1 shows the device utilization summary for the three considered applications when we choose a size of the bus equal to 16 bits. We noticed that for the presented complex DFG, our scheduler uses only 6% of the device slices LUTs which is reasonable. These results are obtained with a design frequency about 19,54 MHz. The device Virtex V XCVLX330 provide 207360 Slices registers, 207360 Slices LUT, 9687 fully used Bit slices, 1200 IOBs, and 32 BUFG/BUFGCTRLs.

These results are confirmed in Figure 16, where we synthesize the same Scheduler for the three applications, but with 10 bits bus size as explained in the following paragraph.

5.3. Accuracy of Execution Times Values. The accuracy of the execution time values is defined by the size of the bus which must convey the information from module to another. The size of this bus is a very determinant parameter in the scheduler synthesis.

Table 2: Behaviour of the scheduler in dynamic situations.

	Total execution time of application	Scheduling time	Size of the scheduler IP	Need to resynthesize
Variation of execution time of tasks	Impacted	Not impacted	Not impacted	No
Variation of partitioning results	Impacted	Impacted	Not impacted	No
Variation of the DFG structure (fork, join, etc.)	Impacted	Impacted	Not impacted	Yes
Variation of the application	Impacted	Impacted	Impacted	Yes
Variation of the execution time precision	Not impacted	Impacted	Impacted	Yes



	Units										
	Master				Slave				RCU		
State	Wait	Ready	Run	Done	Wait	Ready	Run	Done	Wait	Run	Done
S1			1		2				11		
S2				1			2		3,12	11	
S3								2	12,4	3,11	
S4									12,4	11	3
S5	14				13				4	12	11
S6	16 ,5, 15		14				13			4	12
S7	16 ,5, 15		14					13		4	
S8	6	16,15	5	14							4
S9		16,15	6	5					17,7		
S10		15	16	6					17,18 ,8	7	
S11	19		15	16					17,8	7,18	
S12	19		15						8	7	18
S13	9	19	15						17	8	7
S14	9	19	15						17		8
S15	9		19	15					20	17	
S16			9	19	10					17,20	
S17			9		10					17	20
S18			9		10						17
S19				9			10				
S20								10			

Figure 11: Lists of management for a centralised OS.

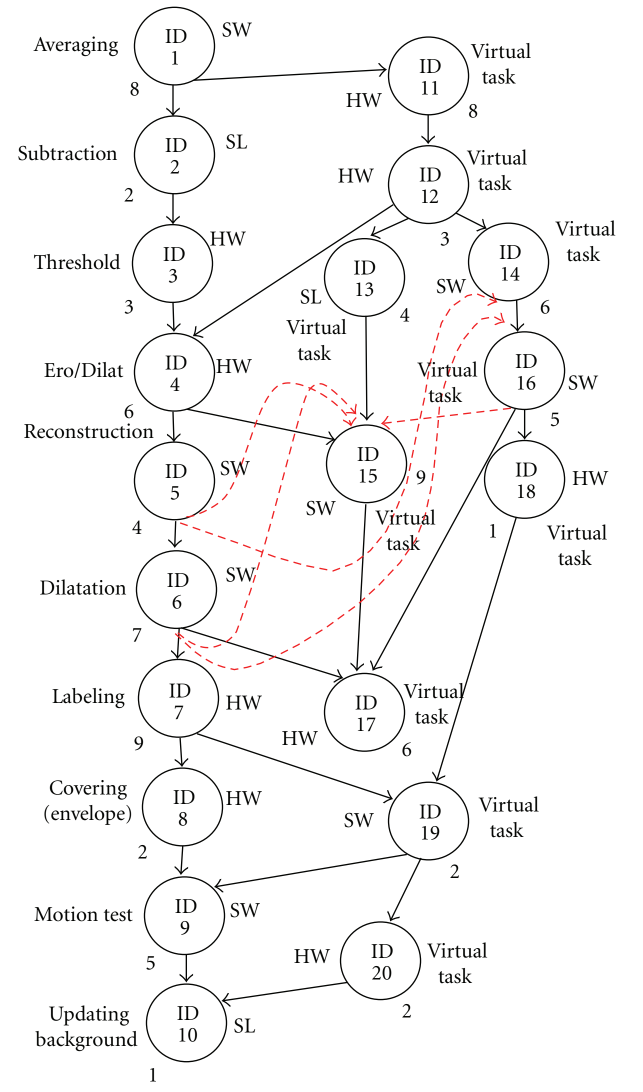


Figure 12: DFG application. The interrupted lines represent the scheduling results.

As shown in Figure 17 , when the size of the bus increases the number of hardware resources used increases also and the frequency of the circuit decreases. But for our scheduler, even with a size of 32-bit, the IP keeps a relatively small size compared to the total number of available resources (20% of Slices LUTs). This is another advantage of our scheduler architecture.

In the general case, the designer has to make a tradeoff between accuracy of performance measures (in this case the execution time) and the cost in number of hardware resources and the maximum frequency of the circuit.

5.4. Summary

5.5. Description of the Scheduling Algorithm. Through the various results of synthesis, we confirm the effectiveness of the hardware architecture for our proposed scheduling method. With these three applications, we have swept most of existing DFGs structures: the sequential in the application icam_simple, the fork and join in the application icam_complex, and the parallelism in the application of robotic vision.

This scheduling heuristic gives better results than the HCP method. Moreover the proposed hardware architecture is very efficient in terms of resources utilization and scheduling latency.

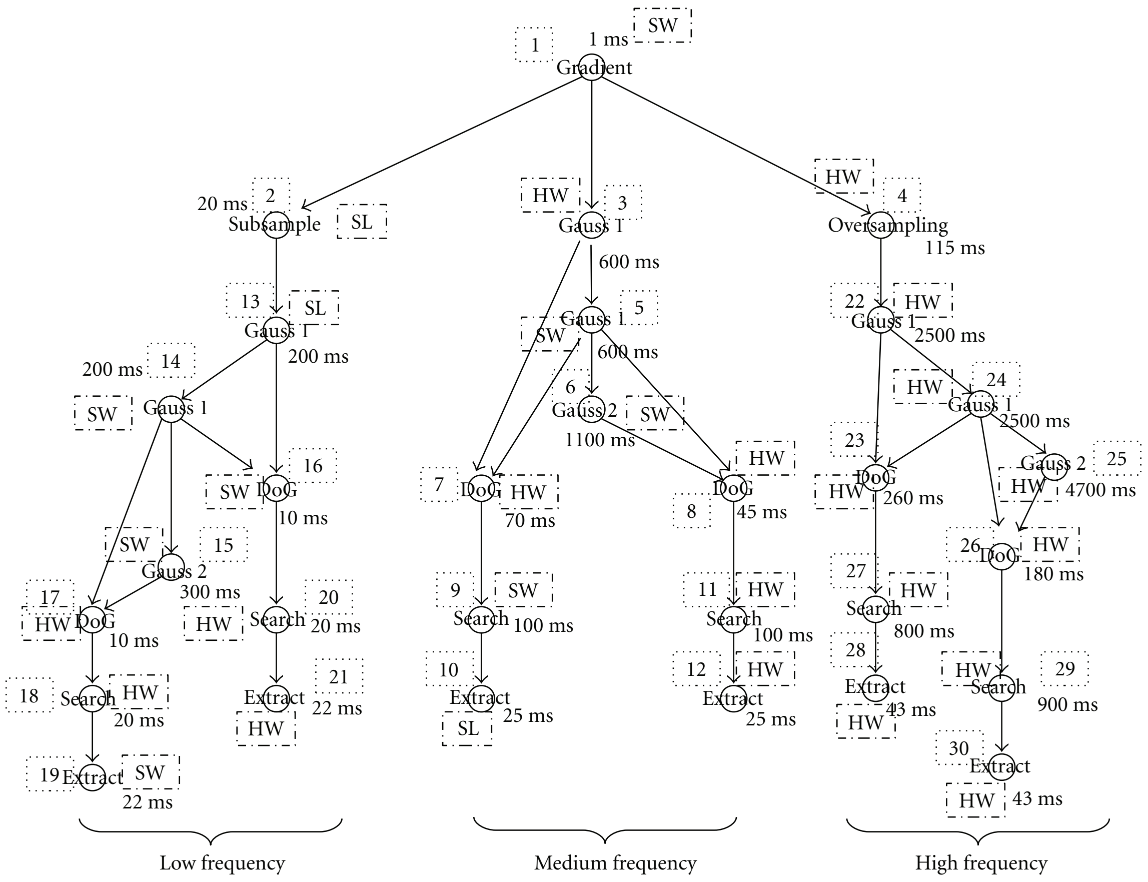


Figure 13: DFG graph of the robotic vision application.

	Icam_simple (10 tasks)	Icame_complex (20 tasks)	Robotic_vision (30 tasks)
Our_total_ execution_time (ms)	47	58	10943
HCP_total_ execution_time (ms)	47	69	10943
SW_scheduling_ time (ms)	7.5	12	21
HW_scheduling_ time (ms)	0.00041764	0.00056274	0.000916976

Figure 14: Execution time for 3 applications.

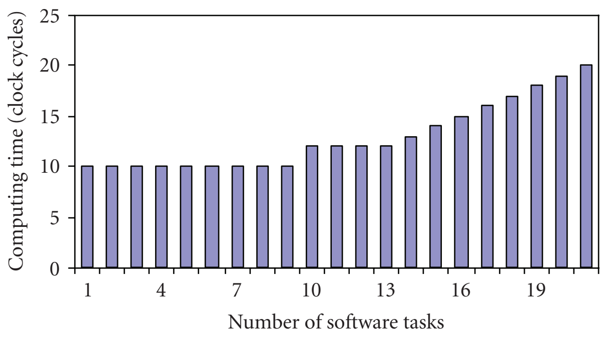


Figure 15: Variation of scheduling computation time according to tasks implentations.

These features allow our scheduling IP to run online and meet the needs of the dynamic nature of most today applications.

6. Conclusions

In this paper, we presented a complete run-time hardware-software scheduling approach. Results of our experiments show the efficiency of the adaptation of the scheduling to a dynamic change of the partitioning that can be due to a new mode of a dynamic application or to fault detection. As developed in this paper, a dynamic HW/SW scheduling approach has many advantages over static traditional approaches. In addition, the efficiency of our hardware implementation gives to our scheduler a minimal overhead in on line execution context.

In conclusion, Table 2 resumes the behavior of our scheduling approach with different situations of dynamicity.

We show through this table in which case it is necessary to restrict the IP scheduler and resynthesize it and in which case the IP can adapt to the dynamic system.

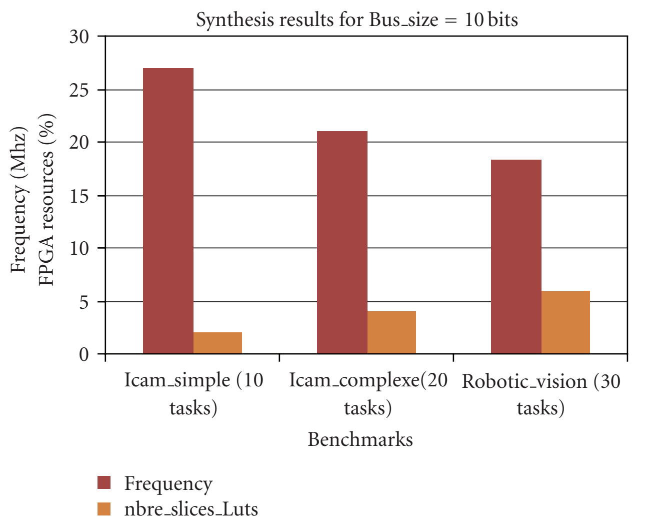


FIGURE 16: Scalability of the method according to application complexity.

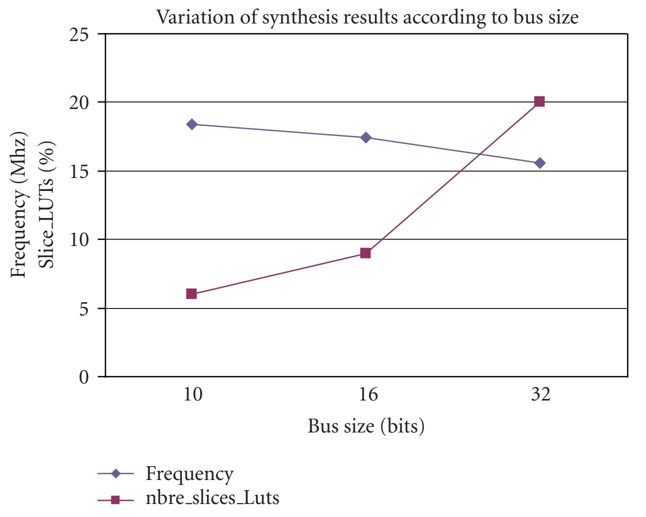


FIGURE 17: Impact of bus size on the scheduler synthesis results for robotic vision application.

Our future works consist in integrating our scheduling approach among the services of an RTOS for dynamically reconfigurable systems.

Appendix

Block Diagrams of the Hardware Scheduler

See Figures 6, 7, and 9.

References

[1] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, Calif, USA, 1979.

[2] Z. A. Mann and A. Orbán, "Optimization problems in system-level synthesis," in *Proceedings of the 3rd Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications*, Tokyo, Japan, 2003.

[3] C. Haubelt, D. Koch, and J. Teich, "Basic OS support for distributed reconfigurable hardware," in *Proceedings of the 3rd and 4th International Workshops on Computer Systems: Architectures, Modeling, and Simulation (SAMOS '04)*, vol. 3133, pp. 30–38, Samos, Greece, July 2004.

[4] J. Noguera and R. M. Badia, "Dynamic runtime HW/SW scheduling techniques for reconfigurable architectures," in *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES '02)*, pp. 205–210, ACM Press, New York, NY, USA, 2002.

[5] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev, "Online task scheduling for the FPGA-based partially reconfigurable systems," in *Proceedings of the 5th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications (ARC '09)*, pp. 216–230, Karlsruhe, Germany, March 2009.

[6] R. Pellizzoni and M. Caccamo, "Real-time management of hardware and software tasks for FPGA-based embedded systems," *IEEE Transactions on Computers*, vol. 56, no. 12, pp. 1666–1680, 2007.

[7] S.-W. Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1215–1227, 2000.

[8] D. Picker and R. Fellman, "A VLSI priority packet queue with inheritance and overwrite," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, no. 2, pp. 245–253, 1995.

[9] B. K. Kim and K. G. Shin, "Scalable hardware earliest-deadline-first scheduler for ATM switching networks," in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pp. 210–218, San Francisco, Calif, USA, December 1997.

[10] T. Pop, P. Pop, P. Eles, and Z. Peng, "Analysis and optimization of hierarchically scheduled multiprocessor embedded systems," *International Journal on Parallel Program*, vol. 36, no. 1, pp. 37–67, 2008.

[11] S. Fekete, J. van der Veen, J. Angermeier, D. Göhringer, M. Majer, and J. Teich, "Scheduling and communication-aware mapping of HW-SW modules for dynamically and partially reconfigurable SoC architectures," in *Proceedings of the Dynamically Reconfigurable Systems Workshop (DRS '07)*, Zürich, Switzerland, March 2007.

[12] B. Miramond and J.-M. Delosme, "Design space exploration for dynamically reconfigurable architectures," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, pp. 366–371, 2005.

[13] Altera Corp., http://www.altera.com/.

[14] Xilinx Corp., http://www.xilinx.com/.

[15] K.M.G. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 579–590, 1999.

[16] J. Resano, D. Mozos, D. Verkest, F. Catthoor, and S. Vernalde, "Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware," in *Proceedings of the 41st Annual Conference on Design Automation (DCA '04)*, pp. 119–124, ACM Press, San Diego, Calif, USA, June 2004.

[17] F. Ghaffari, M. Auguin, M. Abid, and M. B. Jemaa, "Dynamic and on-line design space exploration for reconfigurable

architectures," in *Transactions on High-Performance Embedded Architectures and Compilers I*, P. Stenström, Ed., vol. 4050 of *Lecture Notes in Computer Science*, pp. 179–193, Springer, Berlin, Germany, 2007.

[18] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, Messe, Munich, Germany, March 2003.

[19] F. Verdier, B. Miramond, M. Maillard, E. Huck, and T. Lefebvre, "Using high-level RTOS models for HW/SW embedded architecture exploration: case study on mobile robotic vision," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, Article ID 349465, 2008.

[20] P. Bjorn-Jorgensen and J. Madsen, "Critical path driven cosynthesis for heterogeneous target architectures," in *Proceedings of the 5th International Workshop on Hardware/Software Codesign (CODES/CASHE '97)*, pp. 15–19, Braunschweig, Germany, March 1997.

[21] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

[22] Virtex II Pro, Xilinx Corp., http://www.xilinx.com/.