Hardware-Based Synchronization Framework for Heterogeneous RISC/Coprocessor Architectures

Holger Flatt, Ingo Schmädecke, Michael Kärgel, Holger Blume, and Peter Pirsch
Leibniz Universität Hannover, Institute of Microelectronic Systems
Appelstr. 4, 30167 Hannover, Germany
{flatt,schmaedecke,kaergel,blume,pirsch}@ims.uni-hannover.de

Abstract—**This paper presents a synchronization framework for parallel computing heterogeneous processing elements, which are controlled by a RISC processor. The communication delay between RISC and processing elements is a key issue if the RISC is not closely attached to the processing elements. Recent synchronization approaches neglect communication delays or require low communication delays. This results in a low synchronization rate between RISC and PEs. In order to overcome this delay, a special hardware-based synchronization approach is proposed that reduces the communication overhead and increases the number of executable tasks per time unit. Further, it supports parallel execution of independent hardware tasks. The approach was evaluated for a modular coprocessor architecture containing several processing elements for image processing tasks. The coarse-grained parallel execution of independent tasks significantly improves the speed of an exemplary application for aerial image based vehicle detection on straight highways.**

I. Introduction

In recent years, embedded heterogeneous multiprocessor systems have emerged as a common approach for implementing complex multimedia applications in embedded systems. They are used when computation intensive parts of the application need the performance of a digital signal processor and when a microcontroller is sufficient for control overhead [1].

A common hardware architecture approach for embedded real-time signal processing is the combination of a RISC processor with a coprocessor. As a result of its programmability, a RISC is well suited for execution of high-level algorithms, which contain complex data dependent control structures. RISCs are often integrated into systems-on-chip. Computationally intensive signal processing algorithms are mapped to dedicated processing elements (PEs) inside of the coprocessor. Communication between different PEs is performed via a communication network and memories [2] [3] [4] [5].

If the coprocessor is attached to the auxiliary I/O ports of the RISC, the development of real systems can benefit from available off-the-shelf RISC processors and operating system support. Data transfers between RISC and external coprocessor are relatively slow, so that such an architecture is only suitable for executing applications in which a significant amount of processing can be performed by the coprocessor without RISC synchronization [6].

Signal processing algorithms can be formulated as computation tasks with input and output data. Here tasks are denoted as software tasks if they are mapped to the RISC or as hardware tasks if they are mapped to a specific PE inside the coprocessor. Hardware tasks should be executed without preemption in order to save overhead in terms of hardware cost and context switching time [7]. High system efficiency can be achieved, if the RISC and the coprocessor PEs are always working in parallel. Thus, scheduling mechanisms are necessary, that synchronize the RISC and the PEs inside of the coprocessor. The scheduling must be dynamic in order to allow data dependent control structures in the application.

One approach is a central task scheduler for synchronization. The task scheduling process can be divided into three phases: In the first phase, the task scheduler checks if a resource for processing of the next task is available and if data conflicts are solved. Software tasks can be executed afterwards by the RISC. The hardware task opcode has to be transferred to the PE. In the second phase, the hardware task is handled by the PE. In the third phase, the scheduler has to wait until the PE has finished before assigning a new task. The PE has to send a finish response flag to the scheduler, because the execution time depends on the tasks input data for a variety of algorithms and cannot be predicted therefore.

If scheduling and synchronization are completely implemented in software on the RISC, hardware overhead is kept small. At least, for synchronization of a hardware task two data transfers between RISC and PEs (task start, finish response) are required. In case that the task execution time is much higher than the synchronization time, the communication overhead is small. If the task execution time and its synchronization time are in the same order of magnitude, then the PEs are utilized inefficiently. An example for this behavior is the sequential execution of a high number of tasks on one PE if the execution time of each task is very short.

As a solution, parts of task scheduling, especially synchronization can be relocated into hardware, where fast internal bus systems with low latencies are available. Then, even for short tasks, the synchronization overhead is kept small.

Approaches that shift synchronization tasks into hardware are mainly required for real-time operating systems. The intention is to outsource scheduling algorithms from OS into separate hardware task schedulers [8] [9] [10]. The critical issue is the computation of the task that has to be scheduled next. Since these approaches are scheduling preemptive tasks, they are not suited for scheduling of non-preemptive hardware tasks containing data dependencies.

In [11] the approach of shifting synchronization tasks into hardware was applied to SoCs. Both, software and hardware functions, are modeled as threads. The scheduling hardware comprises several modules and is attached to the system bus. The approach is suited for supporting the RISC with control of a high number of tasks. Operations like thread scheduling or parameter setting require up to 50 clock cycles. The scheduling of software tasks requires a communication structure with low latencies between RISC and hardware. In particular for short tasks, scheduling of both hardware and software tasks causes much traffic between RISC and coprocessor.

A hardware-based scheduling unit in conjunction with a real-time programming model for heterogeneous multiprocessor systems-on-chip is presented in [12]. All processing units, host processors and the scheduling unit are using shared memories. Hardware functions are executed as non-preemptive threads. The scheduling unit maps the tasks to the processing units, controls data dependencies and task priorities. Scheduling of tasks requires approximately 60 clock cycles. The approaches [11] and [12] require more than 50 cycles for task scheduling which limits the efficient execution of short tasks.

This paper presents a new list-based synchronization approach, which provides flexibility for software scheduling while keeping hardware overhead low. Pre-processing of task scheduling is performed in software, which facilitates utilization of arbitrary scheduling algorithms and avoids hardware communication for software tasks. Synchronization and dependency control are swapped out to a centralized hardware-based Dynamic Resource Scheduler unit (DRS). The scheduling approach is embedded into a C++ based application programming framework. An FPGA-based modular coprocessor architecture [5] is used in order to evaluate the efficiency of the proposed Dynamic Resource Scheduler unit.

Throughout the whole paper, an object detection application, which detects vehicles on straight highways in aerial images, is used for illustration and evaluation of the framework [13]. A simplified task graph is shown in Fig. 1, which consists of two parallel sub-graphs. The first one creates a list of all significant lines of the input image. Therefore, a Hough transform and a line detection algorithm are performed. The second sub-graph segments the input image into shapes and provides a list containing vehicle candidates. In the last step, highways are extracted from parallel lines. According to these results, the vehicle list is filtered in order to discard all vehicle candidates outside of the highways. The vehicle filter is mapped to the RISC, whereas all other tasks are mapped to the coprocessor.

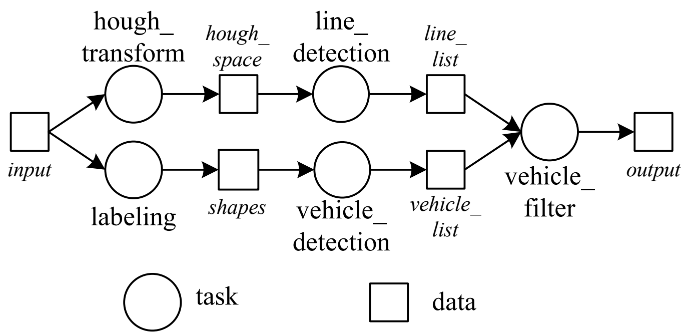

Fig. 1. Task graph of exemplary object detection application

This paper is organized as follows: Section II introduces the list-based synchronization approach. Section III shows the hardware architecture of the proposed Dynamic Resource Scheduler unit in detail. The software interface and programming model are presented in section IV. Results are given in section V. Finally, section VI concludes this paper.

II. List-Based Synchronization

In this section the synchronization rate of a simple RISC system, which has to control several PEs, is introduced. The synchronization rate f_{task} is defined as the number of tasks which can be executed per second. Fig. 2 shows a sequential task graph with n dependent tasks. A task can be started when all preceding tasks with data dependencies have been finished. In terms of execution time, this task graph shows the worst-case scenario due to the sequential nature.

The synchronization rate is calculated for the architectures shown in Fig. 3. First, the RISC performs all of the synchronization functions in software. Subsequently, a Dynamic Resource Scheduler unit is introduced, which is integrated into the coprocessor system.

A linear model is assumed for communication between RISC and coprocessor. The communication time t_c consists of a constant latency $t_{l,RISC}$ for initiating the transfer. This latency includes all overhead caused by the operating system, the memory management unit of the RISC, and several bus systems running at different clock cycles. The data size m of the task opcode, and the bandwidth b_{RISC} of the bus are additional parameters of the linear model. Then t_c is computed as follows:

$$t_{c,RISC} = t_{l,RISC} + \frac{m}{b_{RISC}} \tag{1}$$

A. RISC-Based Synchronization

First, it is assumed that the RISC performs the complete synchronization of the PEs, as shown in Fig. 3 (a). In order to activate a coprocessor PE, the RISC transmits a task call of data size m to the PE. For simplicity, the execution time of the task is consistently denoted by t_{ex}. The time for setting a response flag after the execution of a task is also modeled with $t_{l,RISC}$. Therefore, the whole task execution time for starting, executing and sending a response flag computes to:

$$t_{task,RISC} = 2t_{l,RISC} + \frac{m}{b_{RISC}} + t_{ex} \tag{2}$$

$$\begin{tikzcd}
\text{Start} \arrow[r] & \text{Task 0} \arrow[r] & \text{Task 1} \arrow[r] & \cdots \arrow[r] & \text{Task n-1} \arrow[r] & \text{End}
\end{tikzcd}$$

Fig. 2. Task graph with n interdependent tasks

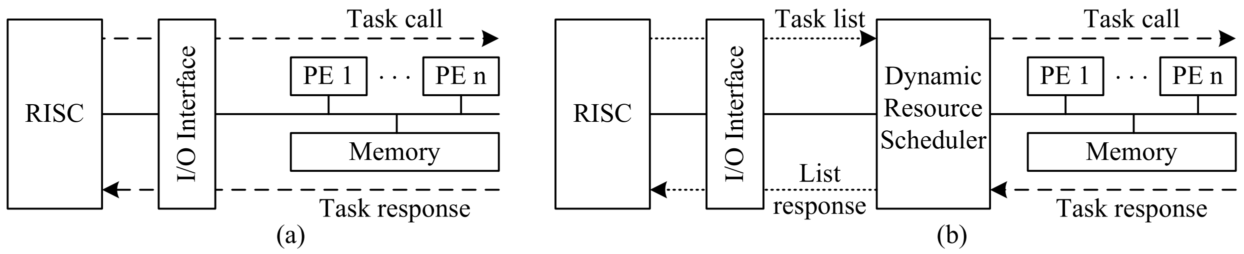

Fig. 3. RISC/coprocessor architecture: (a) Software synchronization, (b) Dynamic Resource Scheduler synchronization

A high communication bandwidth between RISC and coprocessor is assumed. Transferring the task opcode to the PE requires only a few bytes and therefore $t_{l,RISC} >> \frac{m}{b}$ is valid. Thus, (2) is:

$$t_{task,RISC} \approx 2t_{l,RISC} + t_{ex} \tag{3}$$

Thereafter, the maximum synchronization rate computes to:

$$f_{task,RISC} = \frac{1}{t_{task,RISC}} \approx \frac{1}{2t_{l,RISC} + t_{ex}} \tag{4}$$

Eq. (4) shows that for short tasks with $t_{ex} < t_{l,RISC}$ the communication latency $t_{l,RISC}$ limits the maximum synchronization rate.

B. DRS-Based Synchronization

An approach that improves the synchronization rate is list-based synchronization. Instead of synchronizing each task, the opcode of several consecutive tasks is sent as a task list to a Dynamic Resource Scheduler unit, which is integrated into the coprocessor. Fig. 3 (b) shows the resulting architecture. The required time for transmitting a list of n tasks and the response signal after list completion is:

$$t_{c,list} = 2t_{l,RISC} + \frac{n \cdot m}{b_{RISC}} \tag{5}$$

When the DRS receives a task list, it distributes the tasks to the corresponding PEs. Subsequent to the processing of all tasks of a task list, the DRS sends a response signal back to the RISC. The communication structure inside the coprocessor is modeled according to (1) with a constant latency $t_{l,copro}$ and bandwidth b_{copro}. It is assumed that a high speed on-chip communication network between DRS and PEs is available and thus $t_{l,copro} << t_{l,RISC}$ and $b_{copro} > b_{RISC}$ are valid. If the bus constants of the RISC in (2) are replaced by the bus constants of the coprocessor, the execution time $t_{ex,list}$ for a task list with n dependent tasks is:

$$t_{ex,list} = n \cdot (2t_{l,copro} + \frac{m}{b_{copro}} + t_{ex}) \tag{6}$$

The processing time for the list is the sum of (5) and (6):

$$t_{list} = 2t_{l,RISC} + \frac{n \cdot m}{b_{RISC}} + n \cdot (2t_{l,copro} + \frac{m}{b_{copro}} + t_{ex}) \tag{7}$$

Dividing (7) by n yields the processing time of one task $t_{task,DRS}$ when using the Dynamic Resource Scheduler unit:

$$t_{task,DRS} = \frac{2t_{l,RISC}}{n} + \frac{m}{b_{RISC}} + 2t_{l,copro} + \frac{m}{b_{copro}} + t_{ex} \tag{8}$$

The communication time between DRS and PEs can be neglected when it is closely integrated into the coprocessor bus system. Therefore, the bus arbiter has to grant the DRS without much delay in case of bus access conflicts. According to the simplification of (2), the data transmission time $\frac{m}{b_{RISC}}$ can be neglected, too. Subsequently, the synchronization rate $f_{task,DRS}$ is computed to:

$$f_{task,DRS} \approx \frac{1}{\frac{2t_{l,RISC}}{n} + t_{ex}} \tag{9}$$

As shown later for an embedded system, $2t_{l,RISC}$ is equal to 28 μs. This value corresponds to 2800 coprocessor clock cycles if the clock frequency of the coprocessor is equal to 100 MHz. For example, this time is sufficient to process a small region-of-interest (ROI). If the task list contains n similar ROI tasks, the comparison of (4) and (9) shows that the introduction of the Dynamic Resource Scheduler unit reduces the impact of the RISC/coprocessor communication latency approximately by factor n. The value of n has to be chosen carefully, because it affects the complexity for task list generation and hardware cost of the DRS adversely.

III. Dynamic Resource Scheduler

A. Requirements

A dedicated architecture is preferred that requires only a few clock cycles for task synchronization. The main functionality of the DRS consists of receiving task lists and starting tasks on the PEs. A task comprises the following elements:

- PE type
- PE configuration data
- Read addresses
- Write addresses
- Dependencies on other tasks

In order to improve the workloads of the PEs, an out-of-order execution is desired, which requires conflict management. The first type of conflicts are resource conflicts, which have to be solved by the DRS. The second type of conflicts are *Read after write (RAW)*, *Write after read (WAR)*, and *Write* *after write (WAW)* data conflicts. Data conflicts have to be solved by the DRS [14].

The DRS stores the current state of each PE (free, busy) and assigns tasks to free PEs only. The analysis of data conflicts is more complex than the analysis of resource conflicts. It requires information about data dependencies between tasks. For each task in the list, all of the read and write addresses can be compared to all addresses of the previous tasks, which causes significant computational overhead. In order to avoid this high number of comparisons, in software a special data structure for recording data dependencies is used within task list generation. Afterwards, the dependencies on other tasks are transferred within the task list.

B. Hardware Implementation

A dedicated architecture of the proposed DRS [15] that fulfills the requirements above is shown in Fig. 4.

A bus interface connects the DRS to the system bus of the Modular Coprocessor Architecture [5]. It is used for receiving task lists and communication with the PEs.

A double buffering approach is implemented for the task list storage. One task list is executed by the DRS. Parallel to the processing of the first list, the following task list can be uploaded to the shadow memory of the DRS. Hence, the communication delay between RISC and coprocessor can be partially masked. The following list is activated automatically after finishing the first list. Different memory types are used for task list storage. PE type information and task dependencies are stored in internal registers. Therefore, all task conflicts can be analyzed in parallel, which improves synchronization time. PE address and configuration are stored in SRAM, because only data of one task is accessed when the task is started.

The Register File stores all of the status information required during the task list processing. Information about the state (idle, running) of all processing elements and the response configuration are stored in this unit. For task list execution and conflict management, status information for each entry of the task list is provided. Therefore, following states are distinguished:

- No task
- Waiting for execution
- Running
- Finished

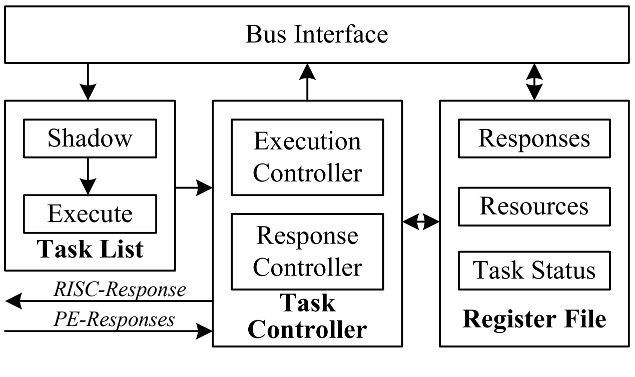

Fig. 4. Architecture of the Dynamic Resource Scheduler unit

The Task Controller is composed of two sub-modules. Before a task can be started, conflicts of all tasks have to be analyzed by the Execution Controller. In order to minimize the required time, all tasks of the list are analyzed in parallel.

Tab. I shows the format of the task list on the basis of the application example. Dependency bits are set for a fixed number of preceding tasks. The tasks *hough_transform* and *labeling* are the first tasks of the task list. Both can be executed immediately, because there are no dependencies on previous tasks. The task *vehicle_detection* depends on the result of the task *labeling*, which is one list position above, and therefore the dependency bit *-1* is set. The task *line_detection* depends on the result of the task *hough_transform*. The dependency bit *-3* is set, because the task *hough_transform* is three list positions above. The DRS starts both tasks *vehicle_detection* and *line_detection* after receiving the finish responses of the tasks *labeling* and *hough_transform*, respectively. For each task, a 1-hot-code is used for coding of the required PEs. Thus, the logic effort of parallel conflict analysis is kept small. The last entry of the task list is empty.

After analyzing conflicts, the Execution Controller selects a task from the list and transmits the task to the associated PE. After task execution, the PE sets its response signal. The response signals of all PEs are simultaneously scanned by the Response Controller, and in case of set signals, the corresponding status flags are updated in the Register File.

In order to improve the flexibility of the DRS, the following basic parameters have to be configured: The maximum length of the task list must be set as trade-off between performance and chip area. Additionally, a dependency parameter can be configured, which allows tasks only dependencies on a fixed number of previous tasks. This reduces the number of possible conflicts that have to be analyzed and the chip area of the Execution Controller. If this number is exceeded by two dependent tasks, special bridging tasks have to be inserted in the task list, which increase the task list size slightly.

Tab. II shows the concept of inserting a bridging task into a task list, which allows only dependencies on two preceding tasks. The bridging task is inserted after the task *labeling* in order to create a dependency between the first and the last task. In this small task list example, the bridging task can be avoided if the task positions of the tasks *vehicle_detection* and *line_detection* are swapped.

TABLE I
TASK LIST FOR DRS UNIT, LENGTH = 5, DEP. DEPTH = 3

Address & Config	Dependencies			Processing Element			
	-1	-2	-3	Hough	Label	V_det	L_det
hough	0	0	0	1	0	0	0
labeling	0	0	0	0	1	0	0
v_det	1	0	0	0	0	1	0
l_det	0	0	1	0	0	0	1
NOP	0	0	0	0	0	0	0

TABLE II
TASK LIST FOR DRS UNIT, LENGTH = 5, DEP. DEPTH = 2

Address & Config	Dependencies -1	Dependencies -2	Processing Element Hough	Label	V_det	L_det
hough	0	0	1	0	0	0
labeling	0	0	0	1	0	0
BRIDGE	0	1	0	0	0	0
v_det	0	1	0	0	1	0
l_det	0	1	0	0	0	1

IV. HW/SW INTERFACE

The HW/SW interface connects the DRS to the application software, running on the RISC. On the one hand, it provides a programming model for simple application development. On the other hand, it executes the application and extracts task lists dynamically and synchronizes with the DRS [16]. The interface is mapped to the RISC.

A. Programming Model

In order to facilitate platform independent application development, it is recommended that the programming model abstracts from hardware. Hence, synchronization mechanisms for task execution and data synchronization between RISC and coprocessor have to be handled by the HW/SW interface at run time. Various data types are used by the tasks, which have to be supported by the synchronization interface. The data must be stored in the RISC memory if used by RISC tasks or in coprocessor memory if used by coprocessor tasks. In order to abstract from hardware, memory allocation for both, RISC and coprocessor memory, has to be done by the interface.

Hence, the HW/SW interface uses uniform data objects for managing task and data synchronization. Memory pointers for RISC and coprocessor memory allow the integration of any data type within the data objects. Furthermore, each data object includes special flags that save the memory type (RISC or coprocessor) in which the data was modified last. According to these flags, the interface allocates memory for the data object and decides whether data transfers between RISC and coprocessor are required or not.

The resulting programming model allows to compose applications by describing program structures sequentially. The system developer has to include information in the static configuration file of the HW/SW interface about tasks that can be executed in hardware. Thus, the HW/SW interface automatically chooses the correct resource and determines task dependencies on the basis of the data objects used by the tasks. A sequential C++ code for the application framework, which implements the exemplary application, is shown in Fig. 5.

```
// Initialize data objects
data_obj input = new_image(WIDTH,HEIGHT);
…
data_obj output = new_vehicle_list(MAX_VEHICLES);

// Application body
hough_transform(input,hough_space);
labeling(input,shapes);
vehicle_detection(shapes, vehicle_list);
line_detection(hough_space,line_list);
vehicle_filter(line_list,vehicle_list, output);
```

Fig. 5. C++ code of exemplary application using data objects

B. Task Execution Control

When application code is executed, initially a common task list for RISC and coprocessor tasks is created. Task entries are buffered for each function. Then the HW/SW interface analyzes the data synchronization flags of the data objects inside the task list for mandatory data transfers between RISC and coprocessor memory. A data transfer has to be performed if a RISC task computes data that is read by a coprocessor task afterwards or vice versa. In order to transfer data, the HW/SW interface adds data transfer tasks to the task list. If mandatory, the Task Execution Control unit allocates both, RISC and coprocessor memory.

If no control structures are inside the application code that depend on data objects, this approach can be used as shown before. A RAW data conflict occurs if the decision of a control structure (if-else, case, for, ...) depends on the result of a data object which will be modified by a task in the current task list. In order to solve this conflict, a special synchronization operation has to be inserted manually or by a parser that stops the generation of further tasks until the data conflict is solved.

Next, the prepared task list is ready for execution. Task processing is managed by a Task Dispatcher module. It is responsible for controlling task execution by considering task dependencies and creating coprocessor task lists for the DRS. The order of task execution is determined by the position within the task list. If the dependency depths of the tasks are exceeding the maximum supported dependency depth of the DRS, then bridging tasks are inserted into the task list.

The Task Dispatcher unit also realizes data synchronization between both processor memory environments and allows parallel task execution by RISC and coprocessor if it is supported by the application.

C. Parallelizing Task Execution

In order to improve system efficiency, the task list can be optimized for parallelizing task execution. Therefore, a Task List Sorting module changes the order of tasks in the common task list before creating a coprocessor task list by the Task Dispatcher. It is intended to group tasks, which have to be processed by the coprocessor. This results in longer coprocessor task lists and the probability of parallel task execution increases.

In addition, RISC tasks are grouped after coprocessor tasks inside the common task list. After transferring the coprocessor task list to the DRS, several RISC tasks follow, which can be executed in parallel to the coprocessor tasks.

The resulting HW/SW Task Scheduling interface is shown in Fig. 6.

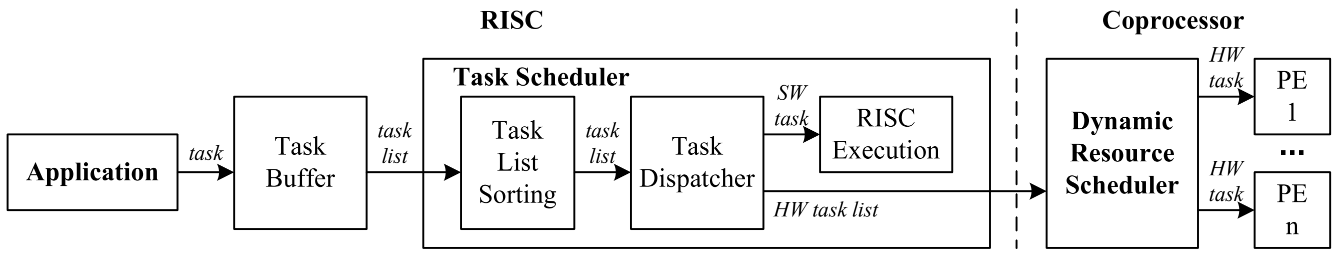

Fig. 6. Architecture of HW/SW Task Scheduling interface

D. Pipelining of Image Processing Applications

If a real-time stream of independent images is processed, the number of independent tasks increases. Therefore, a pipelined processing of consecutive images increases utilization ratio and data throughput of the system. The HW/SW interface as described in the previous sections supports image pipeline only if the application code is expanded. This results in more programming effort and increased code size.

A more convenient approach is outsourcing of the image pipelining functionality into the HW/SW interface. Hence, the HW/SW interface instantiates an application several times by using a thread based solution. Thus, an application is implemented for processing only one image and this single application is used by several threads, which are controlled by the HW/SW interface.

The management of these application threads is realized by a thread scheduler. It is based on round robin scheduling and activates one thread at once. An activated thread is allowed to create a limited number of tasks during thread activation. Afterwards, the thread scheduler chooses the next thread for creating tasks. Overall, the HW/SW interface has access to more independent tasks, which can be used for the creation of longer task lists for the Dynamic Resource Scheduler.

V. Results

A. System Integration

In order to evaluate the hardware based Dynamic Resource Scheduler, it was integrated into a modular coprocessor architecture [5]. The architecture, shown in Fig. 7, comprises a performance optimized 128 bit multi-layer system bus [5] that connects several PEs [17] for computation of image processing algorithms. Internal and external memories are available for data exchange between PEs. A control interface is used for communication between RISC and coprocessor. Synchronization is performed by the DRS, which was attached to the bus system. The coprocessor architecture was mapped to a Xilinx Virtex 5 FPGA. As shown in Tab. III the coprocessor, excluding the DRS unit, occupies approximately 29000 LUTs and 384 KB internal Block RAM. 512 MB DDR RAM and 4 MB SRAM are attached to the external memory interfaces.

B. Dynamic Resource Scheduler

The size of the task list and the maximum dependency depth are the parameters with the most impact on the hardware cost

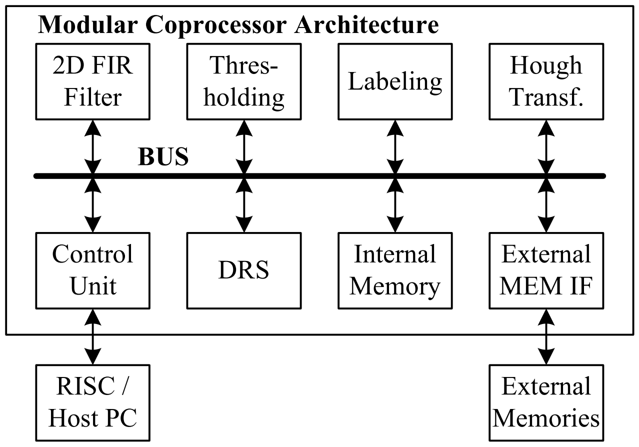

Fig. 7. Architecture for evaluation of DRS and framework

of the DRS unit. Thus, net lists with different configurations were created with Synplify Pro 9.6.2 in order to derive a suitable parameter set. Fig. 8 shows the hardware cost for several task list sizes, which was measured in look-up tables (LUTs). Due to parallel analysis of all tasks, the number of LUTs increases approximately linearly with the task size. For task list sizes up to 32, the hardware cost is just constant with increasing dependency depth, because the dependency check logic is much smaller than the remaining control logic of the DRS.

Thus, the maximum dependency depth can be used and the dependency size check in the HW/SW interface can be switched off, which saves computational effort. For longer task lists, there is a significant impact of the dependency depth

TABLE III
Synthesis results of Modular Coprocessor Architecture

Unit	Look-up Tables	4 KB Block RAMs
Bus System	12719	1
Control	3063	11
Internal Memory / Interfaces	824	9
2D FIR Filter	1976	9
Thresholding	676	4
Labeling	4455	51
Hough Transform	5333	11
Dynamic Resource Scheduler	2638	5
Overall (without DRS)	29046	96
Overall (with DRS)	31684	101

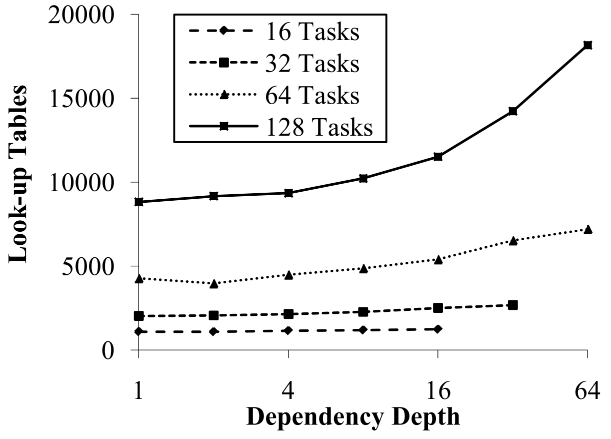

Fig. 8. DRS: Dependency depth over LUTs for different task list sizes

on the hardware costs which suggests to use a configuration with decreased dependency size. For all shown configurations, the clock frequency is higher than 150 MHz and 20 KB Block RAM are required for task storage. 10 clock cycles are required for synchronization of dependent tasks.

A task list size of 32 tasks with maximum dependency depth was used for the evaluation of the framework. It is intended to keep the hardware overhead, caused by the DRS unit and the software overhead, caused by dependency checks, small. In this configuration the DRS unit occupies 2638 LUTs, which is less than 10 % of the whole coprocessor.

C. Synchronization Rate for Short Tasks

Two hardware platforms were used for the evaluation of the framework. A RISC/FPGA-based embedded system was chosen in order to derive bus constants between RISC and coprocessor [18]. The coprocessor was mapped to a Xilinx Virtex 5 FPGA, which is attached to the expansion bus of an Intel IXP460 Network Processor [19]. A communication latency of 14 μs was measured between RISC and coprocessor as shown in Tab. IV. This value is rather high due to the reasons given in section II.

For evaluation of the whole framework, an ASIC emulation system was used [20] instead of the embedded system. The system connects an AMD Athlon 64 X2 4600+ CPU host PC to a Xilinx Virtex 5 FPGA. The C++ application code and the HW/SW interface were mapped to the host PC and the coprocessor to the FPGA which is running at 100 MHz. The bus constants of the Intel IXP460 and the AMD host PC are of the same order of magnitude as shown in Tab. IV.

The synchronization time was measured for short PE tasks with $t_{ex} \approx 0$. Without using the DRS, the processing of short tasks requires 158 μs, which is slightly more than $2t_{l,RISC}$. According to Fig. 2, a task list containing 30 short tasks was built. The introduction of the DRS reduces processing time to 5.3 μs per task which is 1/30 of the value without using DRS.

D. Task Parallelization

In order to evaluate the parallelization features of the programming model, the exemplary object detection application

TABLE IV
MEASURED PARAMETERS OF COMMUNICATION CHANNELS

Device	Frequency	Channel	Latency t_l	Bandwidth b
Intel IXP460	533 MHz	DRS	14 μs	80 MB/s
AMD host PC	2.4 GHZ	DRS	61 μs	39 MB/s
DRS	100 MHz	PEs	30 ns	1.5 GB/s

was used, which utilizes all of the PEs shown in Fig. 7. The coarse-grained task parallelism of the application example was explored for a 16 bit grey scale input picture with 768x576 pixels.

The sequential processing of the sub-graphs {Hough Transformation, Line Detection} and {Labeling, Vehicle Detection} requires 21.79 ms and 12.11 ms respectively. Thus, the theoretical minimum time for parallel execution of both sub-graphs is equal to 21.79 ms. A processing time of 23.76 ms was measured, which is 9 % more than the theoretical minimum. This difference is mainly caused by memory conflicts of several parallel accessing PEs.

Overall, approximately 10 % increased hardware cost facilitate 30 % decreased processing time compared to the sequential execution of both sub-graphs.

VI. Conclusions

In this paper, a hardware-based synchronization framework for heterogeneous RISC/Coprocessor architectures is presented. The framework is optimized for a coprocessor that is connected to the external I/O ports of a RISC.

Contrary to other published frameworks, especially the communication latency between RISC and coprocessor is emphasized. This latency limits the maximum synchronization rate when short coprocessor tasks are executed. The impact of this latency can be considerably decreased by the introduction of the proposed list-based hardware synchronization approach.

The framework can also be applied to SoCs containing an embedded RISC that is closely coupled with the coprocessor. Compared to the system with an external RISC, which was discussed here, the communication latency between the embedded RISC and the coprocessor will be reduced. Therefore, the performance gain of the list-based scheduling approach will be less, too.

Using the example of an object detection application, it is shown that the integration of the approach into a C++ based programming framework allows flexible application development. Furthermore, a parallel execution of independent tasks is done automatically. The synchronization rate for short tasks is significantly improved. For example, image processing applications can be drastically accelerated if a multitude of small region-of-interests have to be processed in hardware.

Although the approach was evaluated by emulation on a host PC connected to an FPGA, these results can also be transferred to embedded systems, because the communication latencies are of the same order of magnitude.

The implemented concept will be applied on more example applications, which contain fine-grained tasks. An essential speedup is expected.

Acknowledgment

The authors thank Steffen Blume and Sebastian Hesselbarth for their contributions to this paper.

References

[1] C. Ravikumar, "Multiprocessor architectures for embedded system-on-chip applications," in *Proceedings of the 17th International Conference on VLSI Design*. IEEE, 2004, pp. 512–519.

[2] M. Arias-Estrada and E. Rodríguez-Palacios, "An FPGA co-processor for real-time visual tracking," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*. Springer, 2002, pp. 710–719.

[3] M.J. Rutten et al., "A heterogeneous multiprocessor architecture for flexible media processing," in *Design & Test of Computers*, vol. 19, no. 4. IEEE, 2002, pp. 39–50.

[4] W. Kim, J.-Y. Chang, and H. Cho, "Pipelined scheduling of functional HW/SW modules for platform-based SoC design," in *ETRI Journal*, vol. 27, no. 5, 2005, pp. 533–538.

[5] H. Flatt, S. Hesselbarth, S. Flügel, and P. Pirsch, "A modular coprocessor architecture for embedded real-time image and video signal processing," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, S. Vassiliadis, Ed., vol. 4599. Springer, 2007, pp. 241–250.

[6] T.J. Todman et al., "Reconfigurable computing: architectures and design methods," in *Computers and Digital Techniques, IEE Proceedings -*, vol. 152, no. 2. IEEE, 2005, pp. 193–207.

[7] S. Jovanovic, C. Tanougast, and S. Weber, "A hardware preemptive multitasking mechanism based on scan-path register structure for FPGA-based reconfigurable systems," in *Second NASA/ESA Conference on Adaptive Hardware and Systems, AHS*. IEEE, 2007, pp. 358–364.

[8] S. Saez, J. Vila, A. Crespo, and A. Garcia, "A hardware scheduler for complex real-time systems," in *Proceedings of the IEEE International Symposium on Industrial Electronics, ISIE*, vol. 1. IEEE, 1999, pp. 43–48.

[9] P. Kuacharoen, M. Shalan, and V. Mooney III, "A configurable hardware scheduler for real-time systems," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2003, pp. 96–101.

[10] J. Hildebrandt, "Hardware basiertes Task-Scheduling für Echtzeit-Systeme," Dissertation (in German), Universität Rostock, Fakultät für Informatik und Elektrotechnik, 2004.

[11] J. Agron et al., "Run-time services for hybrid CPU/FPGA systems on chip," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium, RTSS*. IEEE, 2006, pp. 3–12.

[12] T. Limberg, B. Ristau, and G. Fettweis, "A real-time programming model for heterogeneous MPSoCs," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, M. Berekovic, Ed., vol. 5114. Springer, 2008, pp. 75–84.

[13] H. Flatt, S. Blume, A. Tarnowsky, H. Blume, and P. Pirsch, "Echtzeitfähige Abbildung eines videobasierten Objekterkennungsalgorithmus auf eine modulare Coprozessor-Architektur (in German)," in *ITG Fachtagung für Elektronische Medien Systeme, Technologien, Anwendungen, 13. Dortmunder Fernsehseminar*. VDE-Verlag, 2009.

[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Academic Press, 2006.

[15] M. Kärgel, "Design, VHDL implementation, and verification of a dynamic resource scheduler for an object detection demonstrator," Semester project (in German), Leibniz Universität Hannover, Institute of Microelectronic Systems, 2007.

[16] I. Schmädecke, "Design and implementation of a hardware/software interface for parallel task execution on a modular coprocessor architecture," Diploma thesis (in German), Leibniz Universität Hannover, Institute of Microelectronic Systems, 2008.

[17] H. Flatt, S. Blume, S. Hesselbarth, T. Schünemann, and P. Pirsch, "A parallel hardware architecture for connected component labeling based on fast label merging," in *19th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP*. IEEE, 2008, pp. 144–149.

[18] M. Kärgel, "Design and implementation of an FPGA based evaluation platform for a modular coprocessor architecture," Diploma thesis (in German), Leibniz Universität Hannover, Institute of Microelectronic Systems, 2008.

[19] Intel, "Intel IXP460 network processor," http://www.intel.com/design/network/products/npfamily/ixp460.htm.

[20] ProDesign GmbH, "CHIPit Iridium V5," http://www.uchipit.com/ce/CHIPitIridiumEditionV5.html.