Function-level Multitasking Interface Design in an Embedded Operating System with Reconfigurable Hardware

I-Hsuan Huang, Chih-Chun Wang, Shih-Min Chu, and Cheng-Zen Yang

Department of Computer Science and Engineering
Yuan Ze University, Taiwan, R.O.C.
{ihhuang,ken,csm,czyang}@syslab.cse.yzu.edu.tw

Abstract. Reconfigurable architecture provides a high performance computing paradigm. We can implement the compute-intensive functions into reconfigurable devices to optimize the application performance. In current reconfigurable hardware designs, the function-level reconfigurable hardware has high reusability and low maintenance cost. However, the sharing mechanism and the function invocation interface are still unknown. In this paper, we propose a function-level multitasking interface design to support reconfigurable component sharing in a multitasking embedded operating system. The reconfigurable hardware functions are managed and scheduled by the operating system. Applications can use any needed hardware function via invocation APIs. To study the performance impacts, we implemented a prototype on Altera SOPC development board. We modified μC/OS-II RTOS and evaluated the prototype with prime number search programs and loop programs. The experimental results show the management overhead is acceptable.

Keywords: Reconfigurable computing, multitasking, hardware function, FPGA-based computer, μC/OS

1 Introduction

Reconfigurable computing provides a high performance computing paradigm [3, 4, 9]. In the current development, a general reconfigurable computer comprises one or several traditional microprocessors, and reconfigurable hardware devices. With hardware/software co-design, the reconfigurable computer may execute the compute-intensive tasks on the specific programmable devices. As the reconfigurable hardware can accelerate the task execution, system performance can be highly improved (e.g., [1, 4, 5]).

Adopting reconfigurable hardware has two most prominent benefits. First, overall system productivity can be highly promoted because traditional software-coded functions are accelerated with the reconfigurable computing hardware. Tasks can be thus parallelized with multiple computing engines. Second, the acceleration engine can be flexibly customized due to the reconfigurability. Therefore, the reconfigurable computer can adapt to different computation requirements with high performance.

From the aspect of accelerating granularity, the reconfigurable computing engines can be classified into three categories. The category of the finest accelerating granularity consists of instruction-level processing engines, such as the 2D-VLIW approach [7]. The reconfigurable device operates as a co-processor to execute the task instruction-by-instruction. The instruction-level processing engine approach thus benefits system performance with the expanded instruction set. This approach, however, may incur huge amount of data synchronization overhead between the reconfigurable devices and the general-purpose processors.

The acceleration granularity employed in the second category focuses on the task level [2, 11]. In this task-level acceleration approach, tasks can be either software-coded or implemented as hardware units. Therefore, a software task unit can even have its own correspondent reconfigurable hardware version. While task execution is initiated, the system dynamically decides whether a hardware unit or a software unit is invoked. Since the granularity is at task level, the synchronization overhead between programmable devices and general-purpose processors is highly reduced. Nevertheless, the reusability of hardware task units is very low due to the functional specificity of each unit. Besides, the hardware space efficiency is low because the programmable device needs to maintain all hardware task units in its limited space. The management of hardware task units incurs extra overhead.

The function-level acceleration schemes fall into the third category [6, 10]. The function-level processing engine maintains a consistent hardware interface as the programming interface of software functions. Application developer can follow the programming conventions to use the hardware functions. Due to the high modularity at the function level, hardware function units favor high reusability and low maintenance cost. Although the function-level acceleration approach can fully exploit the high-performance and flexibility of reconfigurable computing architecture, two main issues need to be further discussed: the sharing mechanism of function units and the multitasking interface design. To the best of our survey, previous studies on function-level processing engines mainly focus on performance optimization of specific functions, and rarely discuss the sharing and multitasking issues [6, 10]. Current embedded systems, however, are mostly multitasking systems in which multiple tasks cooperate. To further improve system performance with consideration of the limited space of FPGA, a multitasking interface design providing hardware unit sharing is very crucial.

To support reconfigurable component sharing in a multitasking environment, the enhancements can be practiced in three possible layers: applications, operating system kernel, or reconfigurable hardware. For the following two reasons, we argue that OS kernel support is more superior to other two enhancements. First, if applications take the responsibility to maintain hardware function multitasking and sharing, they need to manage the control registers of hardware functions and maintain function invocations from other applications. Consequently, a task execution may be interfered with other task executions. Application design becomes more complicated and error-prone. Second, if the multitasking mechanism is implemented in reconfigurable hardware, it will occupy a large amount hardware space due to many bookkeeping data structures. Since the space resource is very precious in reconfigurable hardware, this approach incurs high cost/performance ratio in reconfigurable hardware utilization. Implementation of the multitasking mechanism in the OS layer can avoid both the error-prone development problem and the low hardware utilization problem. Although the OS layer enhancement cannot be benefitted from hardware acceleration, its overhead of software-coded execution is comparatively small in the whole system. Accordingly, we propose a function-level multitasking interface design implemented in the OS layer to take advantage of high performance of reconfigurable hardware.

This paper presents the multitasking interface design in an embedded operating system with reconfigurable hardware. The proposed approach has three main design features. First, multiple tasks can coherently share the reconfigurable accelerator hardware with the OS support of the multitasking mechanism. Second, the invocation interface of the hardware units is consistent with the software library to ease application development. Third, if there are numerous tasks waiting for the same shared reconfigurable hardware, OS can dynamically direct the function invocation to the software-coded library. With these features, the system keeps the flexibility to process the application function calls with high performance.

The design of multitasking support for reconfigurable hardware is not straightforward because hardware unit sharing is very different with software library sharing. Two main issues need to be considered: parameter passing flow and data consistency. To maintain data consistency, a management module in OS is designed to deal with multiple invocations, and the OS has a specific job queue to schedule these invocations. In addition, each hardware function unit has its own invocation API in OS to pass parameters. Programmers can replace the compute-intensive function calls in the applications with the corresponding hardware function invocations. When the hardware function unit completes the job, the results are returned via an interrupt service routine. These two issues complicate the design of the multitasking interface.

We have implemented the proposed multitasking interface in an embedded OS μC/OS-II [12] to study the performance impacts. The prototype is based on an Altera SOPC (system-on-programmable-chip) development board [13]. Several hardware functions have been implemented in FPGA to verify the functionality of the multitasking interface. We also conducted preliminary experiments to evaluate the prototype. Although the current benchmark set is primitive, the experimental results show that the management overhead is acceptable and the application performance can be highly improved.

The rest of the paper is organized as follows. Section 2 reviews previous reconfigurable computer studies of different acceleration granularity. Section 3 elaborates the proposed function-level multitasking hardware interface. Section 4 presents the Altera SOPC prototype implementation, and the evaluation results in the experiments. Section 5 concludes the paper.

2 Related Work

The hardware accelerators on FPGA chips are also known as processing engines. From the aspect of accelerating granularity, the processing engines can be classified into three categories, the instruction-level, the task-level, and the function-level. The instruction-level processing engines are usually implemented as co-processors. For example, Santos, Azevedo, and Araujo presented an instruction-level reconfigurable architecture called the 2D-VLIW [7]. In the 2D-VLIW project, the processing engines are controlled by 2D-VLIW instructions which are composed of multiple single operations. The processing engine needs one 2D-VLIW instruction for each execution cycle. Since processing engines are commanded according to 2D-VLIW instructions, the application developer needs to understand the details of the processing engine and to write the 2D-VLIW instructions in the application. Although the processing engine in the 2D-VLIW can be beneficial to applications, the reconfigurable computer needs to synchronize the processing engines and the general purpose processor frequently.

The task-level acceleration approach presents the second category. For example, Andrews et al. proposed the *hthreads* task-level reconfigurable architecture [2]. The *hthreads* is a multithreaded RTOS, which supports the software and the hardware threads it its thread model. Each hardware thread obtains an exclusive processing engine kept in the FPGA chip. The *hthreads* scheduler is responsible for managing the software threads and the hardware threads. Since data of hardware threads are private, the data synchronization can be largely reduced. However, the hardware thread maintenance are cumbersome. Besides, since the hardware thread is customized, the hardware thread can hardly be reused. Accordingly, the utilization of hardware threads is seriously degraded. Another example is the SHUM-μC/OS project proposed by Zhou et al. [11]. Zhou et al. modified the μC/OS-II RTOS and defined their hardware thread model. They implemented a hardware thread control block in the μC/OS-II to keep the data structures of hardware threads. The hardware threads in SHUM-μC/OS are also very difficult to be reused by other applications.

The function-level processing engines are implemented according to the basic algorithmic function blocks of applications. The function-level reconfigurable architecture provides a reusable, elegant, and high maintainability programming paradigm. For example, Rullmann, Siegel, and Merker proposed an application partitioner [6]. The application partitioner extracts compute-intensive algorithmic blocks. Then the compute-intensive algorithmic blocks are implemented as FPGA processing engines. Thus, the system performance can be improved. However, the processing engines are managed by specific applications and cannot be directly used by other applications. Another example is the ReConfigME function-level reconfigurable architecture proposed by Wigley, Kearney, and Jasiunas [10]. Since the processing engine still managed by specific applications, they are very difficult to be reused by other applications. Shibamura et al. also proposed a function-level reconfigurable platform called EXPRESS-1 in 2004 [8]. The major difference between EXPRESS-1 and our system is that EXPRESS-1 focused on the design of the reconfiguration procedure and our system focused on the hardware/software interface design.

3 Functional-level Multitasking Interface Design

The idea of the function-level multitasking interface is to support reconfigurable component sharing in a multitasking reconfigurable computer. Since the processing engines of previous function-level reconfigurable architecture are managed by respective applications, the hardware functions are difficult to be reused by other applications. With the

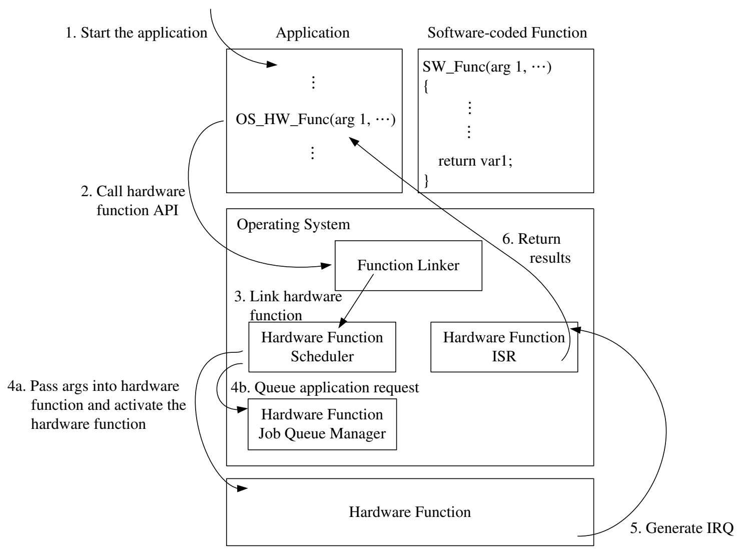
1. Start the application
Application
Software-coded Function
OS_HW_Func(arg 1, ⋯)
SW_Func(arg 1, ⋯)
{
return var1;
}
2. Call hardware function API
Operating System
Function Linker
6. Return results
3. Link hardware function
Hardware Function Scheduler
Hardware Function ISR
4a. Pass args into hardware function and activate the hardware function
4b. Queue application request
Hardware Function Job Queue Manager
Hardware Function
5. Generate IRQ

Fig. 1. Control flow of the hardware function invocation

OS supported function-level multitasking interfaces, all applications can request to use public hardware functions.

Our proposed function-level multitasking architecture modifies four components in a traditional reconfigurable computer: (1) Operating system: We move the management responsibility of hardware functions to the operating systems. We develop a multitasking hardware function manager in the OS. The manager includes a function linker and a scheduler. Besides, we develop a job queue for each hardware function. We provide a multitasking invocation API for each hardware function. (2) Task: The compute-intensive functions can be replaced with the multitasking invocation API. The interface of function parameters and the return results is consistent with the software library to ease application development. (3) Processing engine: In the beginning, each hardware function has to register in the OS. The OS creates the job queue, the function-level multitasking interface, and the interrupt service routine (ISR) for each hardware function. When the hardware function is executing, the hardware function selects new job from the job queue in the OS. When the hardware function finishes a job, it issues an IRQ to the general purpose processor. (4) Software-coded function: Our architecture keeps the software-coded functions. OS can dynamically direct the function invocation to the software-coded library.

Figure 1 shows the control flow of the proposed function-level multitasking architecture. Suppose the hardware function has already been registered. Since the SW_Func() is a compute-intensive function, we implement the hardware version function in FPGA chip. At the same time, we replace the original function call with the hardware function invocation interface called OS_HW_Func(). The control flow is as follows. First, the OS executes the application. Meanwhile, the application is executed by the general purpose

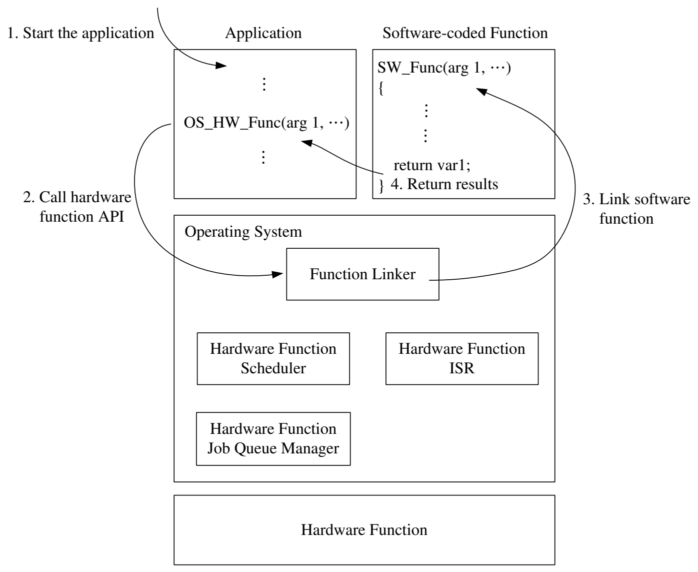

Fig. 2. Control flow when the hardware function job queue is full

processor. Second, the application invokes the hardware function. After the application calls the interface, the OS forces the application enter the waiting queue. Third, After the application passes the parameters to the OS, if the job queue still has free spaces, the function linker forwards the parameters to the hardware function scheduler. Fourth, the hardware function scheduler checks the status of the requested hardware function. If the hardware function is available to be executed, the hardware function scheduler passes the parameters into the hardware function. If the hardware function is busy, the hardware function scheduler saves the hardware function request into the job queue. Fifth, after the hardware function completes one job, the hardware function issues an IRQ. The general purpose processor then jumps to the hardware function ISR to grab computation results of the hardware function. Besides, the ISR configures the hardware function to select next job from the job queue. Sixth, the ISR passes the results to the application. The OS then forces the application enter the ready queue.

Figure 2 presents the control flow when OS dynamically direct the function invocation to the software-coded library. The difference is the third step. When the OS receives the hardware function request, OS jumps to the function pointer of the software-coded function. The parameters are also passed to the software-coded function. Fourth, after the software-coded function finishes the computation, it returns the results to the application. In this case, all operations are executed by the general purpose processor.

Figure 3 shows the hardware function structure used in our architecture. The hardware function is implemented in hardware description language like VHDL or Verilog. In the figure, we denote the hardware function as the processing engine. Each processing engine contains at least four registers, the control register, the status register, the

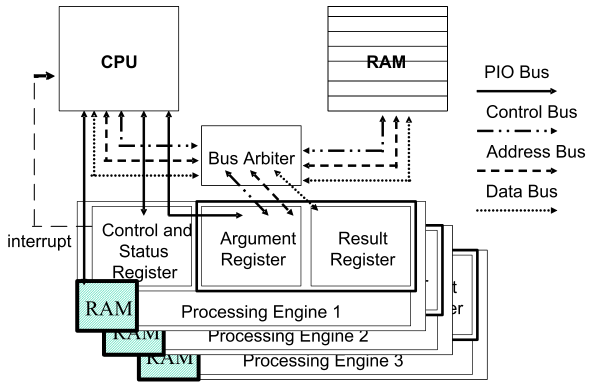

Fig. 3. Hardware function structure

argument register, and the result register. The size of each register is varied according to the application demands. The OS enables and disables each processing engine via the control register. Since the OS has to manage the processing engine, it monitor the condition of processing engine through the status register. The argument register records the parameters of function request. The result register keeps the computation results of the processing engine. Since the parameters and the results may be a pointer variable or a data structure, the processing engine can directly connect to the memory bus. If the parameter is a pointer variable, the processing engine directly access the memory address to capture the variable value. If the processing engine has to access the global variable, it also directly access the memory address of the global variable. When the processing engine completes one job, it issues an interrupt to the general purpose processor. The general purpose processor then executes the ISR to retrieve the computation results. Sometimes, the processing engine needs to call an external function. To solve this issue, the processing engine keeps a private memory space. The memory space contains the function call instructions. When the processing engine wants to call the external function, it first issues an IRQ. The general purpose processor then jump to the ISR. Meanwhile, since the status register shows that the processing engine wants to call an external function, the ISR then creates a new task and jumps to the address of the private memory of the processing engine. After the called function returns the result, rest instructions in the private memory resumes the execution of the processing engine.

4 Prototype Implementation and Experimental Results

We implemented a prototype on the Altera SOPC (system-on-programmable-chip) development board [13] to study the performance impacts. The Altera SOPC development board adopts an FPGA chip to be its core. We programmed the Altera Nios soft IP core processor into the FPGA chip. We choose the μC/OS-II [12] to be our operating system.

To study the performance, we implemented the square root hardware function and the loop hardware function. Both the two functions are compute-intensive. The hard-

```
01 if((in_use->np_piodata & 0x2)){
02
03     ptcb = OSTCBPrioTbl[OSPrioCur];
04     OSTCBCur->OSTCBStat   |= OS_STAT_HDF;
05     OSTCBCur->OSTCBPendTO  = FALSE;
06     hdsqrt_wait++;
07     OS_EventTaskWait(hdsqrt);
08     OS_Sched();
09  }
10  in_use->np_piodata |= 0x2;
11
12  hdsqrt_owner = OSPrioCur; pointer = &hdsqrt_out->np_piodata;
13  *pointer = *parameter; pointer2 = &hdsqrt_in->np_piodata;
14
15  y                       = OSTCBCur->OSTCBY;
16  OSRdyTbl[y]            &= ~OSTCBCur->OSTCBBitX;
17  if (OSRdyTbl[y] == 0) {
18      OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
19  }
20
21  launch->np_piodata |= 0x2;
22
23  OS_Sched();
```

Fig. 4. Implementation of the hardware function scheduler

ware engines are implemented in VHDL and are configured as memory-mapped I/O devices. Each hardware engine needs to be registered in the OS. It also needs to register its interrupt service routine (ISR). For each hardware function, the operating system provides a multitasking interface. Figure 4 shows the implementation of the hardware function scheduler. Taking the square root hardware function as an example, line 1–9 presents the case when the hardware function is busy. We did not really implement a new queue data structure in our prototype, on the contrasty, we adopted the ready table structure in the μC/OS-II kernel. We used the table structure to record applications which is requesting the hardware function. Line 1 checks if the hardware function is busy. Line 3–6 increases the length of waiting queue of the square root hardware function. Line 7–8 sets the application into waiting state and forces the OS execute next application. Line 10–23 presents the parameter passing procedure if the square root hardware engine is available. Line 10 sets the status register of the hardware function. Line 12–13 passes the parameters. Line 15–19 maintains the application state. Line 21 activates the hardware function. Finally, line 23 forces the OS execute next application.

To evaluate our prototype, we run two experiments. In the first experiment, we developed three tasks to search prime numbers. Each task uses the square root hardware function to confirm the prime number. Hence, the square root hardware function is shared by three tasks at the same time. Figure 5 shows the result. The x-axis presents the amount of searched integers. The y-axis presents the processing time. The search workload are shared by three tasks fairly. For example, if we want to find out all prime numbers between 1 to 30000, the task 1 searches 1 to 10000, the task 2 searches 10001 to 20000, and the task 3 searches 20001 to 30000. We compared the processing time of using square root hardware function with the processing time of pure software con-

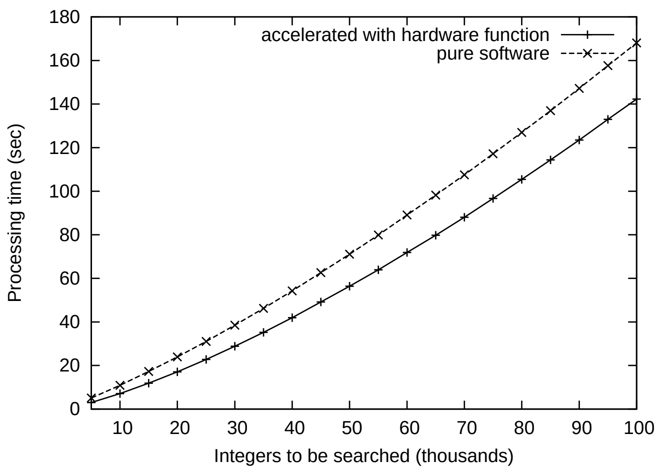

Fig. 5. Performance of the prime number search program

dition. The processing time gains an obvious degradation in 15%. Accordingly, the management overhead of hardware function is little and acceptable.

In the second experiment, we developed a loop program to study the operation overhead of hardware functions. The loop program simply runs "for loop" instructions. Figure 6 shows the result. The x-axis presents the variable n. In each test case, we ran n^3 times loop instructions. The y-axis presents the processing time. The processing time of hardware accelerated configuration is 3–4ms. Hence, the operation overhead of hardware functions is very little and can be ignored.

5 Conclusions

Reconfigurable computing grabs the public attention recently because it provides a high performance computing paradigm. In this paper, we propose a function-level multitasking interface design in an embedded OS to exploit the high-performance benefit of reconfigurable hardware. The proposed mechanism has the following three distinguishing features: (1) multiple tasks can coherently share the reconfigurable accelerator hardware without data inconsistency; (2) the invocation interface of the hardware units is consistent with the API of the software library; (3) OS can dynamically decide whether it resolves the invocation with the hardware unit or with the software library. A prototype implemented with an Altera SOPC development board and μC/OS-II shows its prominent performance improvement in our preliminary experiments. Although the benchmark is still primitive, the positive experimental results convince us of the feasibility in the future development.

In our future plan, the reconfigurable mechanism will be integrated in our prototype. Besides, a more comprehensive benchmark will be implemented to get complete performance characteristics. We also plan to port the proposed multitasking scheme to

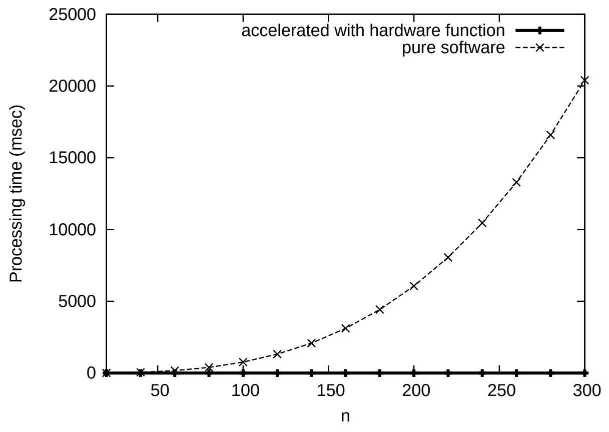

Fig. 6. Performance of the loop program

other famous embedded operating systems, such as uClinux. Improvements on other OS components then will be under investigation to fully take the high-performance benefit of reconfigurable hardware.

References

1. Alam, S. R., et al.: Using FPGA Devices to Accelerate Biomolecular Simulations. IEEE Computer. **40**, 3 (Mar. 2007) 66–73
2. Andrews, D., et al.: hthreads: A Hardware/Software Co-Designed Multithreaded RTOS Kernel. Proc. IEEE ETFA’05. (Sept. 2005) 331–338
3. Buell, D., et al.: Guest Editors’ Introduction: High-Performance Reconfigurable Computing. IEEE Computer. **40**, 3 (Mar. 2007) 23–27
4. Garcia, P., et al.: An Overview of Reconfigurable Hardware in Embedded Systems. Eurasip Journal of Embedded Systems. **2006**, (2006) 1–19
5. Prasanna, V. K., and Morris, G. R.: Sparse Matrix Computations on Reconfigurable Hardware. IEEE Computer. **40**, 3 (Mar. 2007) 58–64
6. Rullmann, M., Siegel, S., and Merker, R.: Optimization of Reconfiguration Overhead by Algorithmic Transformations and Hardware Matching. Proc. IEEE IPDPS’05. (Apr. 2005)
7. Santos, R., Azevedo, R., and Araujo, G.: Exploiting Dynamic Reconfiguration Techniques: The 2D-VLIW Approach. Proc. IEEE IPDPS’06. (Apr. 2006)
8. Shibamura, H., et al.: EXPRESS-1: A Dynamically Reconfigurable Platform using Embedded Processor FPGA. Proc. IEEE ICFPT’04. (Dec. 2004)
9. Todman, T. J., et al.: Reconfigurable Computing: Architectures and Design Methods. IEE Proceedings: Computers and Digital Techniques. **152**, 2 (Mar. 2005) 193–207
10. Wigley, G., Kearney, D.,and Jasiunas, M.: ReConfigME: A Detailed Implementation of an Operating System for Reconfigurable Computing. Proc. IEEE IPDPS’06. (Apr. 2006)
11. Zhou, B., et al.: Reduce SW/HW Migration Efforts by a RTOS in Multi-FPGA Systems. Lecture Notes in Computer Science. **3865** (2006) 636–645
12. Labrosse J.: MicroC/OS-II, CMP Books. (June 2002)
13. Altera corp.: SOPC Builder. http://www.altera.com/products/software/products/sopc/