Architectural Support for Multithreading on Reconfigurable Hardware

Pavel G. Zaykov, Georgi Kuzmanov
{P.G.Zaykov,G.K.Kuzmanov}@tudelft.nl

Computer Engineering Department,
Delft University of Technology
Delft, The Netherlands

Abstract. In this paper, we address organization and management of threads on a multithreading custom computing machine composed by a General Purpose Processor (GPP) and Reconfigurable Co-Processors. Our proposal to improve overall system performance is twofold. First, we provide architectural mechanisms to accelerate applications by supporting computationally intensive kernels with reconfigurable hardware accelerators. Second, we propose an infrastructure capable to facilitate thread management. The latter can be employed by, e.g., RTOS kernel services. Besides the architectural and microarchitecural extensions of the reconfigurable computing system, we also propose a hierarchical programming model. The model supports balanced and performance efficient SW/ HW co-execution of multithreading applications. Our experimental results based on real applications suggest average system speedups between 1.2 and 19.6 times and based on synthetic benchmarks, the achieved speedups are between 1.3 and 29.8 times compared to software only implementations.

1 Introduction

Reconfigurable embedded devices often require multiple applications to be executed concurrently. A common strategy to encapsulate various application functionalities in a conventional software system environment is to use multithreading. Typically, an Operating System (OS) is employed to manage the dynamic creation, execution and termination of multiple threads. If the hardware platform is composed of a reconfigurable logic and a General Purpose Processor (GPP), the OS should be capable to efficiently map the running threads on the available reconfigurable hardware resources.Due to its heterogeneity, the platform complexity and respectively OS service overhead has grown rapidly. As a result, some of the conventional OS kernel services should be optimized to be able to fully exploit the new high performance system capabilities.
The objective of this work is to improve the overall performance of the heterogeneous reconfigurable systems following the multithreading execution paradigm. We provide architectural and microarchitectural mechanisms to accelerate OS kernels and applications in hardware as an extension to the Molen processor [17]. The programming code is organized in a new programming model, which efficiently exploits the proposed hardware architectural and micro-architectural augmentations. The introduced architectural model is not entailed neither to a specific GPP architecture, nor to any reconfigurable fabrication technology. More specifically, the main contributions of this paper are:

– Architectural extensions that allow multithreading applications and RTOS to co-execute in software and in reconfigurable hardware are proposed. More specifically, we extend the processor interrupt system, the register file organization and we modify hardware task synchronization at the instruction level.
– Microarchitectural extensions which support newly introduced Thread Interrupt State Controller (TISC) are provided.
– A hierarchical programming model capable to provide flexible task migration from software to hardware, exploiting inter- and intra-thread parallelism is provided.
– Proposed A Real-Time Interrupt Service Routine (ISR) to support the new Interrupt system is proposed.

Depending on the experimental scenario, results with real applications suggest average system speedups between 1.2 and 19.6 times. Based on synthetic benchmarks, the average speedup is between 1.3 and 29.8 times compared to SW only implementation.
The remainder of the paper is organized as follows. The related work is presented in Section 2. Section 3 describes the architecture and microarchitecture in details - hardware components and interfaces, including XREGs, polymorphic instructions implementation, TISC controller and Interrupt management are presented. Section 4 covers the software prospectives of our proposal, which includes the programming model description. Section 5 provides some specific implementation details and the obtained experimental results. Finally, Section 6 concludes the paper and outlines some future research directions.

2 Related Work

The problem of efficient sharing of hardware computing resources among multiple threads or processes could be solved statically or dynamically. The former involves the usage of advanced compiler techniques and the latter employs an Operating System or a sort of dynamic resource scheduler. The compiler approach solves the resource management problem by performing different optimizations on the application control dataflow graph. Examples of such embedded architectures with static resource management are: MT-ADRES [20] and UltraSonic [7]. In our work, we focus on the infrastructure for dynamic run-time approaches for resource management, therefore we do not address any Compiler related optimizations.
In the dynamic scheduling approach, assuming an RTOS is employed, parts of the programming code, both from the Operating System and/ or the applications, can be transferred onto reconfigurable logic. A detailed classification of the existing reconfigurable multithreading architectures is presented in [21]. Based on it, in the category of dynamic approaches, we identify projects such as [10], [19] and [22]. There, the designers improve the system performance and lower the energy consumption by transferring parts of the OS kernels to hardware. Similar approach is followed by other authors, e.g. [9], [8], where a dedicated hardware resource manager is proposed. The manager takes the decisions using heuristic scheduling and placement algorithms. In contrast to the above cited related works, we propose a system capable to accommodate on reconfigurable logic parts of the user applications, as well as parts of the OS. These OS services could be responsible for the scheduling of hardware tasks but also for the management of

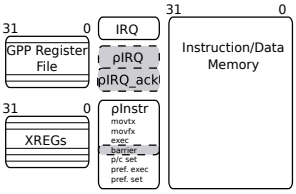

Fig. 1. The Architectural Extensions

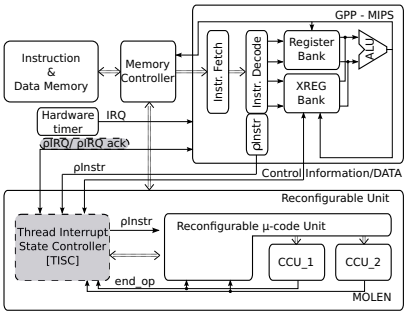

Fig. 2. The Microarchitecture

software tasks. The idea of accelerating OS routines on reconfigurable logic, such as the scheduling of software tasks only, has been already presented in several research projects, e.g. [6]. Our architectural proposal, however, allows to co-execute the management routines for both software and hardware tasks on reconfigurable hardware.

To our best knowledge, the Hthreads [12] is the most relevant project to our current research. The authors use a programming model [1] to distribute the running threads among GPP and reconfigurable logic. The major difference between [12] and our proposal is: We migrate either user application or RTOS thread functions (the concept is described in Section 4) while only complete application threads are moved to hardware in [12]. In [12], the authors use dedicated RTOS modules for communication and synchronization procedures among hardware threads. We believe that our model is more flexible, since it supports migration of parts of the user thread and RTOS kernels in hardware.

There are also projects such as Silicon OS [11], where the Operating System is completely transferred into hardware. Such an approach is not flexible enough and is limited for future improvements, because the RTOS is represented by a complex Finite State Machine (FSM).

The Molen prototype: The Molen Polymorphic Processor [18] consists of a GPP and a Reconfigurable Processor (RP) operating under the processor - co-processor paradigm. The GPP architecture is extended with up to 8 additional instructions, which can support an arbitrary application functionality. Six of these instructions are related to the RP and two - to the parameters transferred between the GPP and the RP through exchange registers (XREGs). The RP related instructions, support different variations of the set-execute paradigm, described in details in [17]. The set-execute model can be supported by an additional “break” instruction, providing synchronization in a sequential consistency programming paradigm. In this paper, we extend the “break” instruction to support synchronization in a multithreading scenario and we call it a “barrier” instruction. In the Molen context, the implementations of application specific functionalities in reconfigurable hardware are called Custom Computing Units (CCUs) and we assumed the same terminology.

In the original Molen design [18] , multithreading was not considered, but in [16], interleaved multithreading (IMT) was addressed and a Hardware scheduler was used

Table 1. Original Molen XREGs

XREG#	31 0
0	CCU Offset
1	Input params, CCU#1
...	...
m	Output param, CCU#1
m+1	Input params, CCU#2
...	...
n	Output param, CCU#2

Table 2. XREGs in Molen TISC Design

XREG#	31 24	23 16	15 8	7 0
0	Input Thread CCU Offset			
1	PID_OUT	TID_OUT	–FREE–	
...	...			
m	PID_IN	TID_IN	FID	Priority
m+1	Input parameter#1 CCU#1			
m+2	Input parameter#2 CCU#1			
m+3	Output parameter CCU#1			

instead of Operating System. The achieved simulation performance speedup reported in [16], with an MJPEG benchmark was 2.75, having a theoretical maximum of 2.78. Since these results were quite appealing, we decided to design a system with an RTOS managing multiple user applications executed concurrently. Moreover, we provide the infrastructure to partition the RTOS and transfer parts of its functionality on reconfigurable logic.

3 Architectural and Microarchitectural Extensions

The proposed **architectural** extensions with respect to [17] are visualized in Figure 1 by shaded blocks. More specifically, they are: 1) new XREG file organization; 2) modified interrupt system, extended with two software accessible registers - ρIRQ/ ρIRQ-ack; 3) modified 'break' instruction, called 'barrier'.

XREG Organization: In Table 1, the original Molen XREG organization is presented. The XREG#0 stores an offset, interpreted as a starting XREG address of the input parameters to the corresponding CCU. In our design, the XREGs are integrated into the GPP core as an extension of the existing register file.

Because of the fact that an RTOS is running concurrently with hardware tasks, some of the CCUs might finish at a time when a different thread has started on the GPP. Therefore, a mechanism is needed that allows the CCU to inform the OS which thread it corresponds to. Another problem occurs, if a context switching is performed after the CCU input parameters and XREG#0 offset are loaded, just before the "execute" instruction is fetched. Later, when the hardware task starts, it might read wrong offset value at XREG#0, if it has been changed by another thread. We solve these problems by: 1) modifying the XREG organization, as suggested by Table 2; and 2) pushing and popping the contents of XREG#0 to/ from the program stack during context switching.

The interpretation of the XREG parameter abbreviations in Table 2 is as follows: Process Identifier (PID_IN), Thread Identifier (TID_IN), Function Identifier (FID) and Priority. Note, the Priority might be equal to the Thread Priority or custom set by the programmer. The FID is used to differentiate multiple hardware tasks executed in task-parallel mode and having the same $\rho\mu$-code address. An example illustrating intra-thread and inter-thread communication is depicted in Figure 5. Tasks f_21 and f_22 are executed on CCU_2, therefore they need to have unique FID. Since both are using the same CCU, they will be consecutively executed according to their assigned Priority. After the CCU computation completes, it writes back the result to an XREG address calculated as the sum of the offset address and the number of input parameters. It also

Table 3. Barrier instruction and its OS interpretation

Barrier instruction				OS interpretation
31 26	25 16	15 8	7 0	
Opcode	FID_Num	FID_1	FID_2	OS_Semaphore_Send
...	...	...	...	OS_Semaphore_Post
FID_N	FID_N+1	FID_N+2	FID_N+3	

writes back its PID and TID to the PID_OUT and TID_OUT fields of XREG#1. They are used by the RTOS to identify which one of the threads is ready for execution.

There is a possibility that multiple CCUs simultaneously acquire read/ write access to the XREGs. The requests are granted according to task Priorities through an XREG Controller, designed as part of the exchange register file. Similarly, we also design an appropriate Memory Controller.

Barrier Instruction: The "barrier" instruction provides synchronization mechanism used by the OS to manage the CCUs execution. It is an extended version of the Molen "break" instruction. In task-sequential execution mode, the barrier instruction participates in each CCU invocation. In task-parallel mode, one barrier instruction corresponds to multiple CCU invocations indicating which of them will be executed in parallel. In Figure 5, the barrier instruction is placed after f_11 in Thread 1, executing it in task-sequential mode and after f_21, f_22 and f_23 in Thread 2 indicating task-parallel mode. An exemplary instruction format of the barrier instruction is presented to the left in Table 3. It has two components - hardware and software representation. The "barrier" instruction is encoded by the "Opcode". The "FID_Num" field corresponds to the number of CCUs synchronized by the current barrier instruction. The multiple "FID_*" fields indicate the Function IDs of the corresponding blocked hardware task. The FID is the unique identifier of the hardware task, as multiple tasks might use the same hardware computing unit.The OS interpretation of the barrier is done by blocking a thread by semaphore. The thread is unblocked only after all CCUs, marked by the barrier, have finished their execution.

Interrupt Handling: In task-sequential execution mode, after a hardware task has completed, it acquires access to the XREGs. When such access is granted, the CCU writes back the computed result in the corresponding XREG. Next, an Interrupt is issued to the RTOS indicating that a task has completed. In task-parallel execution mode, depending on the position of the barrier instruction, CCU could be marked as finished and possibly reused by another task without generating any Interrupt. After an Interrupt is asserted by a CCU, the Interrupt Service Routine (ISR) fetches the content of XREG#1 and unblocks the corresponding thread or Kernel service. Then, the thread is placed in the OS Ready queue and an ρIRQ-ack is send back to signal the TISC.

Microarchitecture: We assume that the GPP has been already extended with the original Molen architecture features and our microarchitectural augmentations are denoted as shaded blocks in Figure 2. The "ρInstr. unit" is a Molen style Arbiter/ Decoder [17], integrated in the GPP Decode stage.

Our main contribution at microarchitecture level is the Thread Interrupt State Controller (TISC). This unit allows concurrent execution of multiple threads having tasks co-executed in software and in hardware. The TISC controller, illustrated in Figure 3,

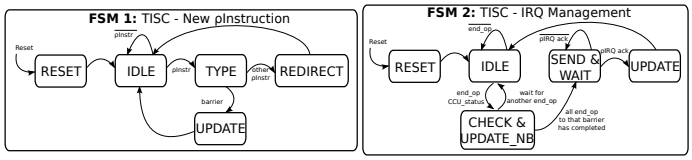

Fig. 3. TISC Finite State Machines

has two finite state machines (FSMs) responsible for instruction predecoding, synchronization and interrupt management of multiple CCUs. The TISC executes the "barrier" instruction at FSM 1: "TISC - New ρ-instruction" in state "Update". The rest of the ρ-instructions are redirected to the Molen Style Coprocessor at FSM 1 in state "Redirect". When a CCU completes, it uses "end_op" signal to inform FSM 2:"TISC - IRQ Management", which switches to "Check & Update_NB" state. The TISC checks whether all "end_op" signals assigned to the corresponding barrier instruction have been activated. If it is the case, FSM 2 asserts Interrupt and jumps to "Send & Wait" state to wait for an 'ρIRQ-ack" signal. When a hardware task completes execution, it generates an interrupt to the processor. The interrupts are consecutively dispatched to the GPP by the TISC Controller according to their priority. Contrary to other approaches found in literature, e.g. [15], which connect each hardware kernel to a separate interrupt vector, we decided to use only one interrupt vector for all active kernels (CCUs). Thus, the achieved system portability is at the cost of minimal time overhead - no more than six additional clock cycles are necessary for the FSMs (see Figure 3). It must be noted that the proposed interrupt mechanism is applicable both in preemptive and non-preemptive execution modes of the CCUs.

4 Software Support

Up to date, there are several widespread multithreading paradigms, such as POSIX Threads [4] and OPENMP [5]. Because of the fact that any of the existing multithreading paradigms for GPP needs to be modified in order to accommodate management for reconfigurable resources, we propose a new hierarchical programming model. The proposed programming model is applicable as an extension to any of the existing standards.

Hierarchical Programming Model: We currently address an embedded system with one GPP core. In order to simplify the software complexity, we partitioned the executed programming code in three abstraction layers - application, thread and task. A small example is illustrated in Figure 4. The application layer accommodates multiple user applications, running independently from each other. Each one of the applications could be composed by one or multiple threads, dynamically created and terminated.

The second level of this abstraction model is the thread layer, where user threads and OS kernel service threads co-exist. At this level only, we positioned control and data dependencies between the threads. In the example of Figure 4, we assume that Application 1 has two threads - Thread A and B, which are communicating between each other.

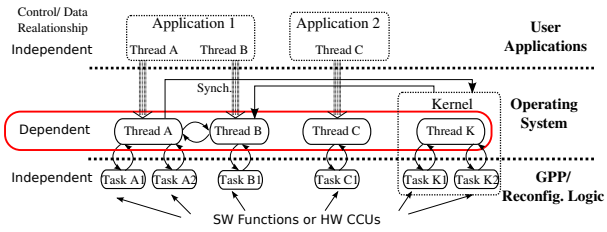

Fig. 4. The proposed Hierarchical Programming Model: An Example

In our programming model, the communication/ synchronization channel is established through the OS, encapsulated in tasks K1 and K2.

The source code of a user thread contains one or multiple tasks. These tasks are the building blocks of the third layer. Depending on where a task is executed, we distinguish two types of the tasks: a function and a CCU. When a task is executed in software, we refer to it as to a function; when a task is executed in reconfigurable hardware, we refer to it as to a CCU. All tasks have the following property - when started, they do not communicate with each other, i.e., they do not contain neither control nor data dependencies. In software, a task, being a function, receives a set of input parameters, performs computations and returns a result. These input parameters are transferred through the GPP Registers and the program stack. In hardware, the task input parameters are transferred through a preassigned exchange registers, described in Section 3, and a special "execute" instruction is invoked to start the execution. When the CCU completes, it writes back the computed result to a dedicated exchange register or in a designated location in shared memory.

Intra-thread and Inter-thread parallelism: To simplify the scenario, we assume that the RTOS is running only on the GPP, scheduling two user threads - Thread 1 and Thread 2, depicted in Figure 5. Each thread is composed by multiple tasks, some executed on CCUs. In Thread 1, it is f_11 running on CCU_1 and in Thread 2, tasks f_21, f_22 are executed on CCU_2 and f_23 is running on CCU_3. An example of software executed task/ function is f_12 from Thread 1. The time slots during which the thread is running on reconfigurable logic are marked by solid lines. The thread execution time on the GPP is denoted by a dashed line. The thick solid line marks the time when Thread 1 is blocked during synchronization/ communication period with Thread 2.

The programming model supports two levels of parallelism - intra- and inter-thread parallelism corresponding to two execution modes - task-sequential and task-parallel. The type of the execution mode is corresponding to the location of the special "barrier" instruction in the programming code.

Task-sequential mode addresses Inter-thread parallelism - In this mode, each CCU is executed sequentially. When it is finished - it signals back the processor, and the next thread continues its execution. An example is task f_11 from Thread 1 in Figure 5.

Task-parallel mode addresses Intra-thread parallelism - In this mode, multiple hardware CCUs and/ or software functions could be co-executed in parallel. In Figure 5, such tasks are f_21, f_22 and f_23 from Thread 2. The concurrent execution of CCUs inside of a single thread mimics the traditional out-of-order execution. The CCU synchroniza-

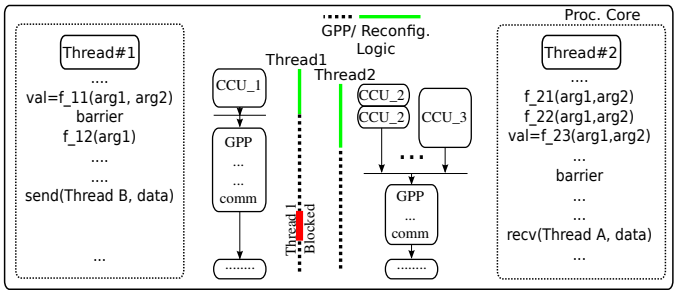

Fig. 5. Inter- and Intra-thread Parallelism: An Example

tion is controller at the software level by a dedicated barrier instruction, described in more details in Section 3.

Figure 5 also visualizes a scenario when the system has to execute CCUs acquiring more reconfigurable resources than the available ones. For example, in Thread 2 - tasks f_21 and f_22 are using the same hardware representations (CCUs). The only differences are the values of their input parameters. As it is assumed, multiple CCUs can be consequently executed over the same hardware. In current implementation, we assumed that all CCU resource requests will be always fulfilled.

5 Experimental Setup and Results

We have developed an experimental platform based on Xilinx Virtex II XC2VP30 FPGA chip using the XUPV2P Prototyping Board. In the following, we briefly discuss our experimental setup.

GPP Core: The GPP, used to obtain the experiments, has a traditional RISC architecture. It is based on the MIPS R3000 [13] implemented as a soft-core on the FPGA chip.

RTOS: The RTOS running on the GPP is a light-weight version of [13], which we named ρRTOS. It has a tiny memory footprint, less than 20 KBytes, with a support of multithreading, memory management, synchronization and round-robin (RR) scheduling.

To satisfy our RTOS requirements, we made the following modifications to [13]: 1) improve the ISR that services the hardware timer - the XREG#0 value is pushed and popped to/ from the program stack; 2) a new ISR is designed, managing thread semaphores; 3) a Molen programming library is used to emulate the Molen instructions.

Compiler Support: We do not address any static scheduling techniques by the Compiler. We create the MIPS compatible binary code, by using a standard version of the GCC compiler. Instead of modifying the compiler to consider Molen polymorphic instructions, we create a Molen programming library which emulates them. In our experimental implementation, the FIDs are statically generated by the system programmer, but ultimately they should be managed by an appropriately designed Compiler.

Evaluation Methodology: We run several real streaming applications and we design our own synthetic benchmark suite to evaluate the impact of the RTOS and the Interrupt

Table 4. Evaluation Results with Floyd Algorithms

	Scenarios						
Applic. Type:	Pure SW		Reconfig. HW		SW + Reconfig. HW		
	1SW1T	4SW4T	1C1T	4C4T	3C1SW4T	2C2SW4T	1C3SW4T
FL25	47 650	120 483	46 520	112 079	116 497	116 981	119 382
FL400	158 082	568 505	50 830	126 787	240 635	346 849	454 976
FL1600	914 349	3 591 126	65 470	187 630	1 032 139	1 886 204	2 739 194
S_{av}(Var. Floyd)	1	1	6.1	8.2	2.3	1.5	1.2

Controller on the thread and task performance. Each one of the experiments includes: thread creation, thread termination, interrupt handling and an OS scheduling policy algorithm. To evaluate the system performance in scenario *k*, we use average speedup, denoted by $S_{av}(k)$:

$$S_{av}(k) = \sum_{i=1}^{n\_exp} \frac{T_{SWe}(i)}{T_{HWe}(i) * n\_exp} \quad (1)$$

$T_{SWe}(i)$ corresponds to the computation time of the software only implementation and $T_{HWe}(i)$ is the computation time in each one of the other scenarios, containing reconfigurable hardware executions. The *n_exp* variable represents the number of performed simulations in each one of the scenarios.

We employ the following nomenclature to structure the Scenario names(S_N): S_N= $\{S_N$,(DL)$\}$ where D=$\{$1,2,3,4$\}$ and L=$\{$C,SW,T$\}$. S_N is composed by multiple (DL) couples, where *L* are interpreted as follows: *C* corresponds to CCU, *SW* is Software task and *T* is Thread. For example: 4C1T should be interpreted as 4 CCUs running in parallel in 1 thread; 1C3SW4T means 4 threads, one executing CCU, the others - running in software. Note, that in the 4C1T scenario, the TISC Unit shall assert an interrupt to signal the GPP only after all four hardware tasks (CCUs) are finished.

The streaming package includes three popular applications: *Floyd-Warshall* algorithm, *Conjugate Gradient* and *MJPEG* Encoder. We choose them, because they present three different application domains: graph analysis, linear equation systems and multimedia domain. The results, in terms of clock cycles, are presented in Table 4. Note, that in any other scenario than 1C1T/ 1SW1T, there are four CCUs, identical to the one used in 1C1T/ 1SW1T. The CCUs (e.g 4C4T) are executed in four different threads, each one working on dataset sizes equal to the one from 1C1T/1SW1T.

Real Benchmarks: The *Floyd-Warshall* algorithm (FL) finds all shortest paths in a weighted graph. In Table 4, it is marked as FL25, FL400 and FL1600, where the numbers are the count of nodes in the graph. Working with small data-sets - FL25, the execution time of the system among all scenarios is almost equal to the pure software execution time - 1SW1T and 4SW4T. The reason for such a behaviour is caused by the OS overhead in terms of thread_creation and scheduling routines. Working with larger datasets, FL400 and FL1600, the execution time of 1C1T/4C4T remains relatively constant compared to pure software. The experiments composed by software functions and hardware CCU threads such as 2C2SW4T, mimics the behaviour of 4SW4T due to the software tasks. The FL CCU is designed following the implementation details given in [3].

The second experimental application is based on the *Conjugate Gradient* (CG) benchmark, part of the *NAS* Parallel Benchmark Suite [2]. The most computation intensive

Table 5. Evaluation Results with CG and MJPEG applications

Applic. Type:	Scenarios		S_{av}
	Pure SW 1SW1T	Reconfig. HW 1C1T	
CG14	72 251 488	3 684 817	19.6
MJPEG64	4 030 275	1 269 830	3.2

parts of this application are the floating point arithmetic operations. As results suggest, even with small number of trails - 14, running such applications on a simple RISC core without floating point unit using software math library only, consumes tremendous amount of time. This is the reason that we do not perform any experiments with larger datasets and we do not run more than one thread. The purpose of this benchmark is to indicate the potential portability of our ideas in application domains traditionally positioned outside of embedded world. On the other side, we demonstrate that we can port such complex applications in embedded systems, that have not been considered before. The experimental results, reported in Table 5, suggest acceleration of more than 19 times compared to the pure software implementation. The experiments are produced, using a dedicated memory hierarchy which efficiently feeds the CCUs with data. The description of such a new hierarchy is outside the scope of this paper. The reason of such high acceleration ratio is the fact that more than 95% of the application computation time is spend in a simple function. More implementation details of the CG CCU could be found in [14].

The most time intensive function of the *MJPEG* Encoder, we considered, is the Discrete cosine transformation (DCT), which we implemented in a CCU. The experimental results, reported in Table 5, suggest that the overall application execution time drops more than 3 times for a tiny video stream with 64 pixels (8×8) per frame.

Synthetic Benchmarks: Last but not least, we have designed a synthetic benchmark suite, which covers more use-cases than the previously described real applications. The suite is predominantly composed by arithmetic and/or logical operations with limited number of memory accesses. The experimental results of our synthetic benchmarks are visualized in Table 6. They include two basic scenarios: 1) software functions are executed 10 times slower than their corresponding hardware implementations; and 2) when software functions are 100 times slower than the corresponding CCUs. The execution time of the CCU, modelled as number of iterations in a single loop, varies from 100 cycles upto 12000 clock cycles. Depending on the hardware acceleration ratio (10 times or 100 times), the number of software executions varies from 10×100 upto 10×12000 and from 100×100 upto 100×12000. All synthetic simulations are implemented with four tasks, executed over variable number of software threads with equal priorities.

The experimental results in 4C4T and 4C1T have almost constant execution time while the dataset size has been scaled. The execution time difference of almost four times between 4C4T and 4C1T is caused by an OS overhead. In use-cases such as 3C1SW4T and 2C2SW4T with dataset size equal to 200 elements, the system performance is even lower than pure software implementation - 4SW4T. The performance degradation is caused by the applied Round Robin scheduling policy which further delays ready for execution software threads composed by hardware tasks. An appropriate dynamic pri-

Table 6. Experimental Results with our Synthetic Benchmark Suite

	SW tasks are running 10x slower than HW implementation						SW tasks are running 100x slower than HW implementation					
Dataset	4C1T	4C4T	3C1SW4T	2C2SW4T	1C3SW4T	4SW4T	4C1T	4C4T	3C1SW4T	2C2SW4T	1C3SW4T	4SW4T
100	22 865	90 951	92 449	92 538	96 532	197 852	22 865	90 951	112 035	133 891	155 597	177 229
200	22 965	90 940	93 114	97 136	101 382	167 335	22 965	90 940	439 304	314 000	241 231	297 986
500	23 015	92 469	97 461	113 940	121 646	133 243	23 015	92 469	210 234	297 342	425 868	530 231
1000	23 226	92 035	111 874	133 857	155 597	178 239	23 226	92 035	306 114	538 857	764 621	990 551
2000	24 312	93 982	436 304	314 000	234 015	297 986	24 312	93 982	479 348	1 323 123	1 122 233	1 893 149
4000	25 807	94 625	176 648	269 143	358 575	448 696	25 807	94 625	953 265	1 884 584	2 033 432	3 698 131
8000	26 968	96 541	263 322	312 540	689 630	810 119	26 968	96 541	1 544 811	2 382 857	3 523 423	7 223 112
10000	27 465	98 135	306 037	538 857	764 132	990 551	27 465	98 135	1 629 822	3 383 123	4 128 654	9 123 232
12000	29 969	100 112	354 556	618 788	867 546	1 170 848	29 969	100 112	1 796 858	3 996 828	4 696 858	11 488 934
S_{av}	11.5	2.7	1.7	1.6	1.3	1	29.8	7.3	2.3	1.7	1.5	1

ority scheme could potentially solve this problem.

6 Conclusions and Future Work

In this paper, we proposed a holistic architectural support for performance efficient multithreading execution on reconfigurable hardware. More specifically, a new programming model for inter- and intra-thread parallelism was introduced and several architectural and microarchitectural improvements were proposed. The system was verified by means of synthetic benchmarks as well as by real applications. In order to benefit from our Custom Computing Machine model, a system programmer should consider the following recommendations: First, restructure the programming code and employ CCUs in task-parallel execution mode whenever possible - especially with tasks having short execution times. Second, carefully select the number and type of threads working on CCUs, since such an action can even decrease the system performance if applied improperly. If the system has hard realtime requirements, more advanced scheduling algorithms should be employed. In our future works, we shall focus on transferring parts of the RTOS on hardware. Looking at further prospective, we believe that our ideas are widely applicable on heterogeneous and homogeneous multiprocessor platforms, as well.

7 Acknowledgement

This work is carried out under the COMCAS project (CA501), a project labelled within the framework of CATRENE, the EUREKA cluster for Application and Technology Research in Europe on NanoElectronics, the HiPEAC European Network of Excellence - cluster 1200 (FP6-Contract number IST-004408) and the Dutch Technology Foundation STW, applied science division of NWO (project DSC.7533).

References

[1] Andrews, D., Niehaus, D., Jidin, R., Finley, M., Peck, W., Frisbie, M., Ortiz, J., Komp, E., Ashenden, P.: Programming models for hybrid FPGA-CPU computational components: a missing link. IEEE Micro 24, 42–53 (2004)

[2] Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The nas parallel benchmarks. Tech. rep., The International Journal of Supercomputer Applications (1991)
[3] Bondhugula, U., Devulapalli, A., Fernando, J., Wyckoff, P., Sadayappan, P.: Parallel fpga-based all-pairs shortest-paths in a directed graph. In: IPDPS (2006)
[4] Buttlar, D., Farrell, J., Nichols, B.: PThreads Programming: A POSIX Standard for Better Multiprocessing. O'Reilly Media (1996)
[5] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel programming in OpenMP. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
[6] Chandra, S., Regazzoni, F., Lajolo, M.: Hardware/software partitioning of operating systems: a behavioral synthesis approach. In: GLSVLSI '06. pp. 324–329. ACM (2006)
[7] Haynes, S.D., Epsom, H.G., Cooper, R.J., McAlpine, P.L.: UltraSONIC: A reconfigurable architecture for video image processing. In: FPL'02. pp. 482–491. Springer-Verlag (2002)
[8] J.Cui, Q.Deng, He, X., Z.Gu: An efficient algorithm for online management of 2D area of partially reconfigurable FPGAs. In: DATE. pp. 129–134 (2007)
[9] Marconi, T., Bertels, K., Lu, Y., Gaydadjiev, G.: Online hardware task scheduling and placement algorithm on partially reconfigurable devices. In: ARCS. pp. 306–311 (2008)
[10] Marescaux, T., Nollet, V., Mignolet, J.Y., Bartic, A., Moffat, W., Avasare, P., Coene, P., Verkest, D., Vernalde, S., Lauwereins, R.: Run-time support for heterogeneous multitasking on reconfigurable SoCs. Integration 38(1), 107–130 (2004)
[11] Nakano, T., Utama, A., Itabashi, M., Shiomi, A., Imai, M.: Harware implementation of a real-time operating system. In: 12th TRON Project International Symposium. pp. 34–42 (1995)
[12] Peck, W., Anderson, E., Agron, J., Stevens, J., Baijot, F., Andrews, D.: HTHREADS: a computational model for reconfigurable devices. In: FPL. pp. 885–888 (2006)
[13] Rhoads, S.: http://www.opencores.org/project,plasma
[14] Roldao, A., Constantinides, G.A.: A high throughput fpga-based floating point conjugate gradient implementation for dense matrices. ACM Trans. Reconfigurable Technol. Syst. 3(1), 1–19 (2010)
[15] Tumeo, A., Branca, M., Camerini, L., Monchiero, M., Palermo, G., Ferrandi, F., Sciuto, D.: An interrupt controller for fpga-based multiprocessors. In: ICSAMOS. pp. 82–87 (2007)
[16] Uhrig, S., Maier, S., Kuzmanov, G.K., Ungerer, T.: Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling. In: RAW. pp. 209–217 (2006)
[17] Vassiliadis, S., Wong, S., Cotofana, S.D.: The MOLEN $\mu\rho$-coded processor. In: (FPL), Springer-Verlag (LNCS) Vol. 2147. pp. 275–285 (August 2001)
[18] Vassiliadis, S., Wong, S., Gaydadjiev, G.N., Bertels, K., Kuzmanov, G.K., Panainte, E.M.: The Molen polymorphic processor. IEEE Transactions on Computers 53, 1363–1375 (November 2004)
[19] Walder, H., Platzner, M.: Reconfigurable hardware Operating Systems: From design concepts to realizations. In: Engineering of Reconfigurable Systems and Algorithms. pp. 284–287. CSREA Press (2003)
[20] Wu, K., Kanstein, A., Madsen, J., Berekovic, M.: MT-ADRES: Multithreading on coarse-grained reconfigurable architecture. In: ARC. LNCS, vol. 4419, pp. 26–38. Springer (2007)
[21] Zaykov, P.G., Kuzmanov, G.K., Gaydadjiev, G.N.: Reconfigurable multithreading architectures: A survey. In: SAMOS-IW. pp. 263–274 (July 2009)
[22] Zhou, B., Qui, W., Peng, C.L.: An operating system framework for reconfigurable systems. In: CIT. pp. 781–787 (2005)