COMPILER ASSISTED RUNTIME TASK SCHEDULING ON A RECONFIGURABLE COMPUTER

Mojtaba Sabeghi, Vlad-Mihai Sima, Koen Bertels[*]

Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
email: {sabeghi, vladms, koen}@ce.et.tudelft.nl

ABSTRACT

Multitasking reconfigurable computers with one or more reconfigurable processors are being used increasingly during the past few years. One of the major challenges in such systems is the scheduling and allocation of the tasks on the reconfigurable fabric. In this paper we present a two level scheduling mechanism for tightly coupled reconfigurable architecture machines. To overcome the complexity of identifying kernels at runtime, we use the compiler support. The compiler provides the runtime system with a configuration call graph which will be used as a viable source of information for the scheduling algorithm. We combine the configuration call graphs from all running applications and extract the distance to the next call and frequency of calls in future for each kernel from this graph. We base our scheduling decisions on these two parameters. Evaluation results show that the proposed method is very promising and it has the potential to be considered for future research.

1. INTRODUCTION

Reconfigurable architectures have received growing interests for the design of application specific computer systems in the past few years because of their adaptability and short design time. That is mainly because they can increase the performance with accelerated hardware execution, while possessing the flexibility of a software solution. In such systems, selected computation intensive parts of the application can be mapped onto the reconfigurable hardware which results in faster execution.

However, due to the fact that currently the compiler and other design time tools deal with the mapping at compile time and not at runtime, the design of a reconfigurable system is not as flexible as it could be [1, 2, 5, 6]. Most of the current trends have assumed only a single thread of execution where the given application has full ownership of both the host microprocessor and the reconfigurable processors. Even in case of multiple applications, the scheduling is static as they assume the designer knows everything about the application execution at design time. Nevertheless, with reconfigurable computers containing FPGAs with millions of system gates, it is now feasible to consider the possibility of serving multiple arbitrarily concurrent applications on a high performance reconfigurable machine.

Dynamic and partial reconfigurations are important factors in sharing the FPGA logic area among competing applications. Run-time reconfiguration provides the ability to change the configuration not only between applications, but also within a single application.

The problem here is to assign the limited system resources based on the needs of the applications at the execution time. Compton et al in [7] proposed a knapsack algorithm based scheduler that views the system behavior as a series of static knapsack problems, solved at specific intervals. Furthermore, they assume that the behavior and resource requirements of the system are known at the scheduling points.

To solve the same problem for Image Processing Pipelines, Authors in [8], proposed exhaustive search, integer linear programming and local search methods for pipeline assignment. There are also another category of runtime task scheduling mechanism in which the system assumes the hardware implementation of the tasks are in rectangular shapes and are relocatable [9, 10, 11].

However, in this paper, we present a scheduling approach in which we put no limitation on the size and the shape of the hardware kernels. The scheduling points are not predetermined and are dependent on the dynamic of the system. Furthermore, we will not use sophisticate methods, which increase the time complexity of the system.

In [3], the authors present an algorithm for periodic real time tasks which can be preempted. Our algorithm tackles a slightly different problem, as tasks are not periodic, not real time and cannot be preempted. Compared to [4] our scheduling algorithm can schedule a task in software, which will help in high system load scenarios.

We are targeting the tightly coupled processor coprocessor MOLEN paradigm. The MOLEN polymorphic processor [18] consists of a general-purpose processor

* This research has been funded by the hArtes project EU-IST-035143, the Morpheus project EU-IST-027342 and the Rcosy Progress project DES-6392

(GPP) tightly coupled with a reconfigurable coprocessor. The latter can be used to implement arbitrary functions in hardware using custom computing units (CCUs).

The rest of the paper is organized as follows. In section 2, we describe the MOLEN programming paradigm and the execution environment. Section 3 presents our scheduling algorithms and section 4 includes the performance evaluation results. We conclude the paper in section 5.

2. BACKGROUND OVERVIEW

2.1. Molen Hardware Organization

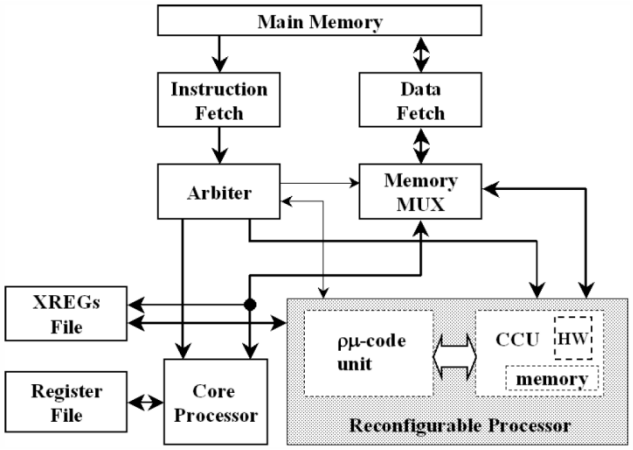

Fig. 1. Molen Hardware Organization

The programming model for a reconfigurable platform must offer an abstraction of the available resources to the programmer, together with a model of interaction between the components. The MOLEN programming paradigm [12] abstracts the hardware and allows the programmer, the compiler and the runtime system to efficiently use the underlying hardware.

Molen is established based on the tightly coupled co-processor architectural paradigm. Within the Molen concept, a general-purpose core processor controls the execution and reconfiguration of reconfigurable coprocessors (RP), tuning the latter to various application specific algorithms. A task, executed by the RP, is divided into two distinct phases: *set* and *execute*. In the set phase, the RP is configured to perform the required task and in the execute phase the actual execution of the task is performed. This decoupling allows the set phase to be scheduled well ahead of the execute phase, thereby hiding the reconfiguration latency.

The Molen hardware organization, depicted in figure 1, consists of two parts; the general-purpose processor (GPP) and the reconfigurable processor (RP) usually implemented on an FPGA. The Exchange Registers (XREGs) are used for data communication between the Core Processor and Reconfigurable Processor.

2.2. The Runtime Environment

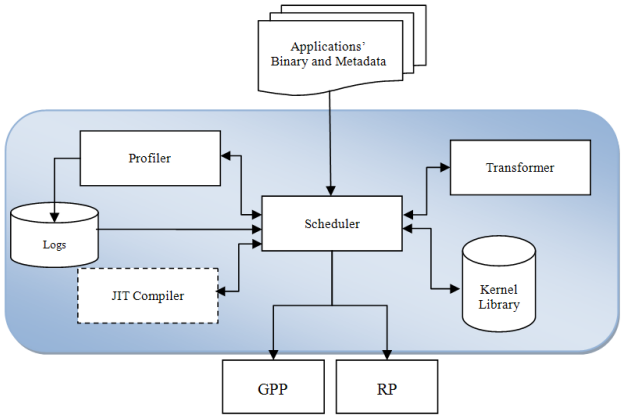

Fig. 2. The Runtime Environment

Our runtime environment which is presented in [13] is in fact a virtualized interface, which decides how to allocate the hardware at run-time based on the dynamic changing conditions of the system. Moreover, this layer hides all platform dependent details and provides a transparent application development process. This layer is located above the Operating System.

Here, we present a brief overview of the components in figure 2. More details of the environment are outside the scope of this paper. The focus of this paper is mainly on the scheduler.

2.2.1. The Scheduler

The goal is to end up with certain parts that can be accelerated on the RP and the remaining parts of the application will be executed on a regular general-purpose processor. A 'part' of the application can be a whole function or procedure or even any cluster of instructions that is scattered throughout the application. The term part here is equal to the term task.

Task scheduling can have different conflicting objectives, like improving performance, power consumption or the memory footprint. As a result, many different scheduling algorithms can be used here. For example during the program execution, an scheduler can estimate the potential speedup that will be achieved when running the task on reconfigurable hardware and estimate the initial cost of a hardware mapping. Then, based on the scheduling policy and considering the other applications requirements, the scheduler decides how to allocate the hardware.

2.2.2. The Profiler

The Profiler continually tracks the application behavior and records statistics such as the number of references to one task. These statistics when combined with the scheduling policy, are used to determine where, when and how to execute the tasks (GPP or RP). The profiler must log the collected data in very fast and efficient data structures from

which the data can be retrieved with no burden to the system performance.

2.2.3. The Transformer

When the scheduler decides to run a task in hardware, the transformer has to replace the software implementation of the task with a call to the hardware implementation. In fact, binary rewriting imposes a considerable overhead to the system and to avoid that overhead we use the MOLEN programming paradigm in which the compiler puts the set and the execute instructions besides the software version of the kernel in a conditional structure. The runtime system sets a conditional variable based on the scheduling decision for each kernel.

2.2.4. Kernel Library

The kernel library is a precompiled set of tasks for a specific architecture. For each task multiple versions can exist, each with different characteristics. These will be saved as metadata and can contain, among other things, the configuration latency, execution time, memory bandwidth requirements, power consumption and physical location on the reconfigurable fabric.

2.2.5. JIT Compiler

A just-in-time compiler can be used to compile the tasks for which there is no implementation in the library. The compiler converts binary to bit stream. Any just in time compiler can be used here nevertheless there is no efficient one yet available. Warp processors [19] is a project which uses JIT FPGA compilation.

3. THE SCHEDULING ALGORITHM

One of the basic limitations of multi-tasking reconfigurable computers is the size of the reconfigurable fabric. The size of the required logic for all the applications usually exceeds the size of the fabric. Therefore, the most important design factor for a runtime scheduling mechanism is the replacement policy. This replacement algorithm determines which part of the logic area should be replaced whenever some space is needed. Of course, it is preferable to replace the block that has the least chance of being referenced soon. In the simplest scenario, the replacement policy assumes the hardware tasks are rectangularly shaped and relocatable. In this case, the problem is to arrange those rectangles in an optimized way. However, with the current technology status, these two assumptions are not realistic.

It could be a nice attempt to reduce latency by guessing which tasks are likely to be needed soon and either pre-configuring such tasks or if they are already configured, not to replace them with other tasks. This is often in combination with pre-cleaning, which guesses which configured tasks currently on the FPGA are not likely to be needed soon. However, this cannot be managed by the runtime systems because, it is impossible to compute reliably how long it takes before a task will be used again, except when all the applications running on a system is known beforehand.

One solution is to provide information from the design time. Our compiler helps the runtime scheduler by generating the Configuration Call Graph (CCG).

3.1. Configuration Call Graph

The Configuration Call Graph is a directed graph, which provides the runtime system with information about the execution of the application. Each CCG node represents a task. The edges of the graph represent the dependencies between the configurations within the application. The nodes of the CCG are of three types: operation information nodes, parallel execution nodes (AND nodes), alternative execution nodes (OR nodes).

The parallel execution nodes specify that all of their successor nodes have to be executed, respectively configured, in parallel. The alternative execution nodes specify that only one of their successors have to be executed. However, the selection of the successor depends on a condition. The output edges of the alternative execution nodes are weighted with the probabilities of execution of the corresponding successors. The compiler derives those probabilities from the edge execution frequencies. The weights of the other edges are set to one. Figure 3 shows a sample CCG.

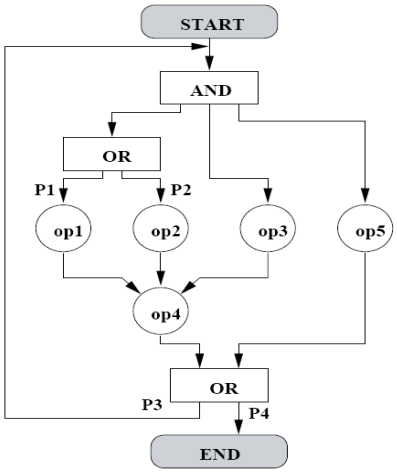

Fig. 3. A Sample CCG

Using the CCG, we can look at future and extract useful information such as the *distance to the next call* and the *frequency of the calls in future* which can be used as the replacement decision parameters.

3.2. The Replacement Policy

Many decision parameters can be used to decide for the replacement such as the frequency of use in the past, the distance from last call, or even a random selection.

Furthermore, other parameters like the area fragmentation can be also considered.

All of the aforementioned parameters are heuristics to keep the best kernel configured. Having information about the future is indeed a real success factor in this decision. In this paper, we introduce two decision parameters: the *distance to the next call* and the *frequency of the calls in future* which can be obtained from the CCG. Furthermore, one can dig a bit more and decide on more complicated parameters like the *expected speedup* or *maximum attainable speedup*.

In this paper, we focus on the distance to the next call and the frequency of the calls in future. We explain how we calculate these two parameters along these lines.

At each scheduling point, we define the distance to the next call as follows. For application i, let d_i be the distance to the next call. This represents the number of task calls between the current execution point and the next call to the task T, in the breadth first traverse of the CCG. We use the minimum value of d_i in our algorithm as the distance to the next call.

Similarly, for a specific task T, we define the frequency of the calls in future for an application as the total number of calls to the task T in all the successors of the current execution point. Afterwards, the frequency of the calls for the task T in the whole system is the sum of the frequencies of all applications.

It should be mentioned that in our traversal algorithms for calculation of the distance and frequency, for the alternative execution nodes (OR nodes), we traverse the node with the highest probability. Furthermore, in case of coming across a loop, we will traverse the loop only one more time (two times in total).

Deciding based on the distance to the next call means we want to remove the task which will be used furthest in future. Similarly, considering the frequency of the calls in future as the decision parameter means the replacement candidate is the task which will be used less frequent in the future.

It might seem that the replacement here is very similar to the page replacement algorithms for virtual memory managements. For example, recency of use and frequency of use are the most dominant decision factors for virtual memory page replacement algorithms. However, there are some major differences here; firstly, in virtual memory replacements when the memory is full, the replacement algorithm has to replace at least one page by a new page. Nevertheless, here, evicting no kernel is also an option. Second, the cost of the replacement in virtual memory is fixed and has no effect on the decision yet here every kernel has its own configuration latency. Third, most of the algorithms for virtual memory are based on the temporal and spatial locality in the reference pattern of the main memory. However; in our case, the spatial and temporal locality of references is not yet proven or accepted as a general principle and needs more investigation. Lastly, a virtual memory page is relocatable and can be placed everywhere in the memory, nonetheless, this is not true for the hardware kernels.

3.3. Compiler assistance

Intercepting the kernels at run time is not trivial. It requires complex runtime profiling tools, which might impose a considerable overhead. However, without losing the generality of the runtime system, we can use the compiler hints for determining the tasks. The compiler at compile time generates the instrumented binary based on the design time profiling information. Within the Molen programming paradigm, the set and execute instructions can be effectively employed as a mean for instrumentation. The runtime scheduler then considers the Molen executes instruction as starting point of the tasks.

Furthermore, the compiler performs static scheduling of the reconfiguration requests assuming single application execution. The goal of this scheduler is to minimize the impact of the reconfiguration latency over the application performance. The reconfiguration latency is an important drawback in the reconfigurable computers. However, by configuring the tasks well in advance and not sooner than necessary, we can reduce the effect of reconfiguration on the overall application performance. More information about the compiler optimization can be found in [14].

Additionally, the compiler generates the configuration call graph (CCG) from which we can predict the future behavior of the system.

3.4. The algorithm

The scheduling procedure is a two level mechanism. The first phase is a normal scheduling policy for example a normal round robin. At the first level of scheduling, the tasks are being scheduled to run on the general-purpose processor. Apart from that, there are some points that the system needs to decide to use the reconfigurable coprocessor for computation intensive tasks. These points are either the set/execute instructions or the internal events from the runtime profiler.

At each scheduling points, there are a few possibilities. The first choice is to do nothing and continue the execution on the general-purpose processor. The other option is to choose one of the hardware implementations of the kernel from the library, configure it on the FPGA and start hardware execution. Remember that there might be other conflicting tasks already configured on the FPGA and some of them might be in the execution status (busy).

Let us assume we are at a certain scheduling point. The scheduler is going to decide about the task T. There are n different hardware implementations (I_1, I_2 … I_n) in the library, matching T.

List 1. The scheduling procedure at a certain scheduling point deciding about hardware/software execution of task T

```
1- Scheduling begins
2- Assume implementation_list is a list of all the possible
   implementations of T;
3- For each I_i in the implementation_list, check if the corresponding
   physical location is busy. If it is busy (executing), remove I_i from
   the list;
4- Assume AlreadyConfigured_list is an empty list;
5- For each I_i in the implementation_list, check if it is already
   configured, and if so add it to the AlreadyConfigured_list;
6- If AlreadyConfigured_list is not empty
      6-1- For all I_i in the AlreadyConfigured_list, find I* with the
           minimum execution time;
      6-2- Start I* execution;
      6-3- GOTO 9 ;
7- If implementation_list is not empty
      7-1- For all I_i in the implementation_list, find I* with the
           minimum (configuration_latency + execution_time)
      7-2- Remove I* from the implementation_list;
      7-3- Assume ToBeEvicted_list is an empty list;
      7-4- Add all the kernels which are already configured on the
           FPGA and have overlaps with I* physical location to the
           ToBeEvicted_list;
      7-5- For all I_j in the ToBeEvicted_list, if there is a I' with a
           distance to the next call smaller than the distance to the
           next call of I* go to 7;
      7-6- Start I* execution;
      7-7- GOTO 9 ;
8- Continue with software execution
9- Scheduling ends
```

First, the scheduler must ensure that in the physical location of each I_i on the FPGA, there is no other busy task configured. Afterwards, it has to check if any of I_i is already configured and is in ready status. This means there is no need for reconfiguration and the hardware execution can start right away. Subsequently, the scheduler must choose either to replace one or more of the currently configured tasks on the FPGA with one of the I_i or to continue the T's execution in software. This can be done using the replacement policy. Unsurprisingly, there is a chance that no replacement takes place because it might be more efficient to keep the currently configured tasks. In this case, T has to run in software. Listing 1 presents the scheduling procedure in more details.

Considering steps 6-1 and 7-1, it is clear that we are optimizing the execution time. As we mentioned before, other optimization can be also applied such as selecting the smaller task in size or the task with less energy consumption and so on. The replacement decision is being taken in step 7-5. In the listing 1, we used the distance to the next call as our decision parameter.

By looking more carefully at the scheduling procedure, it is obvious that the hardware execution part is non-preemptive. This is why before any replacement decision we check the busy status of the physical locations in step 3.

When switching to preemptive hardware scheduling, the system has to offer, among other things, the ability to save and restore hardware status. In [16, 17] more information about task preemption can be found. Preemptive scheduling is outside the scope of this paper.

The overhead of the algorithm in listing 1 is very dependent on the number of implementations for each task because a number of constraints have to be checked for each of them. The distance to the next call and frequency of calls in future can be calculated offline for each application and at runtime we only need to find the minimum value of the distances for all the applications and to sum the frequencies for all of them. So, this will not impose a big overhead.

4. PERFORMANCE EVALUATION

In this section, we compare three algorithms which are based on the distance to the next call, frequency of calls in the future and frequency of calls in the past. The frequency of calls in the past has been proposed in [7] with the name *Most Frequently Used (MFU)*. In our implementation of this algorithm, the RC scheduling intervals are not fixed from before and we run the algorithm at each of our scheduling points.

We performed simulations in order to evaluate performance of the presented algorithms. We use our discrete event simulator which is an extension of the CPUSS CPU scheduling framework [20]. Our target architecture is similar to MOLEN hardware platform on Xilinx Virtex series. We assume there is enough local memory on the FPGA so that the kernels can be executed independently without bus conflicts. The simulation workloads are generated after carefully analysing various hardware kernels. It includes short life-time, medium life-time and long life-time tasks as well as small size, medium size and large size tasks. All the measures are based on the numbers in the benchmark model presented in [15]. To get an impression of the FPGA size, in the worse case, the FPGA can accommodate only one large size task based on our measures and in the best case it can accommodate around 80 small size tasks.

In our workloads, the hardware implementation of a task is in average 5 times faster than the software implementation. Each application consists of 50 tasks including redundant tasks (an application can have between 5 to 20 unique tasks). The tasks can be shared between applications. We assume each task has at least one hardware implementation. In average each task has three different versions of hardware implementation.

It should be mentioned here that the data dependencies between the tasks within one application should be managed inside the application and the runtime system can not do anything about it. In fact, it is the application that sends the requests for task execution and the request should not be issued unless all the dependencies are met.

Tables 1 and 2 show the obtained results for each algorithm. The last column is the software only execution of the tasks. Figure 4 shows the execution times in case of 12 applications (600 tasks). Both the numbers in the table 1 and figure 3 certify that the distance to the next call is a better parameter to decide on. It means we replaced the tasks that are less likely to be referenced soon.

Table 1. The execution time for tasks execution (ms)

Number of Applications (Number of tasks)	*Distance to the next call*	*Frequency in future*	*Past Frequency (MFU)*	**Software Only**
3 (150)	**148620**	**151913**	**171065**	**467104**
6 (300)	**245137**	**254336**	**298587**	**684642**
12 (600)	**691755**	**731469**	**805506**	**1324149**
60 (3000)	**3066189**	**3605675**	**4207550**	**7006287**

Table 2. The percentage from the software execution time

Number of Applications (Number of tasks)	*Distance to the next call*	*Frequency in future*	*Past Frequency (MFU)*	**Software Only**
3 (150)	**32%**	**33%**	**37%**	**100%**
6 (300)	**36%**	**37%**	**44%**	**100%**
12 (600)	**52%**	**55%**	**61%**	**100%**
60 (3000)	**44%**	**51%**	**60%**	**100%**

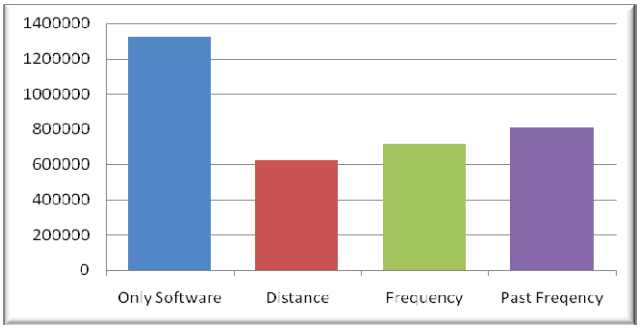

Fig. 4. The execution times for 12 applications (ms)

Additionally, the results show that the frequency of calls does not perform as well as the distance to the next call. The reason is that although a kernel might be used several times in the future, it does not mean that those uses are in the near future. For example, a task might be accessed 10 times during 10 seconds from which one access is in the first one second and the rest are in the last 2 seconds. Using highest frequency algorithm suggests keeping this task on the FPGA, however seven seconds between the calls might be enough time to remove this task from the FPGA and bring it back later. Despite this fact, we believe that the frequency is a good decision parameter especially for the workloads with periodic execution behaviour. The reason for better performance of future frequency over the past frequency is also obvious. Looking at the past frequency is a kind of heuristic to predict the future and it is not always accurate. Using the CCG in the future frequency algorithm, we have a better heuristic with more accuracy.

Considering overall application performance compared to the software only execution, we obtained speedup of 2.29. As mentioned before the hardware implementations of the task are, on average, 5 times faster. The past frequency algorithm also obtained speedup of 1.66. In the very ideal case with our assumption, the speedup should be around 5 if all the tasks run in hardware.

5. CONCLUSION AND FUTURE WORKS

In this paper we proposed a scheduling mechanism which is based on the compile time information prepared by the compiler and presented in the form of a graph to the runtime system. This scheduler is a part of our runtime environment which is responsible for allocating the reconfigurable hardware at run-time and hiding the platform dependent details from the programmers. In the future, we will try to further optimize our scheduling decisions. For example, considering the FPGA area fragmentation, may increase the resource utilization and as a result the overall performance. In addition we will investigate other replacement parameters such as the maximum attainable speedup. Furthermore, we will investigate other design factor like the power consumption.

6. REFERENCES

[1] J.M.P. Cardoso and H.C. Neto, "Compilation for FPGA-Based Reconfigurable Computing Systems", *IEEE Design and Test of Computers*, vol. 20, no. 2, Mar./Apr. 2003.

[2] G. Dimitroulakos, N. Kostaras, M.D. Galanis and C.E. Goutis, "Compiler assisted architectural exploration for coarse grained reconfigurable arrays", *In Proceedings of the 17th ACM Great Lakes Symposium on VLSI,* Stresa-Lago Maggiore, Italy, March 11 - 13, 2007.

[3] A. Ahmadinia, C. Bobda, D. Koch, M. Majer, J. Teich,"Task scheduling for heterogeneous reconfigurable computers", Proceedings of the 17th symposium on Integrated circuits and system design, September 2004

[4] Juanjo Noguera, Rosa M. Badia, "Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling", *ACM Transactions on Embedded Computing Systems* (TECS), Vol 3, Issue 2, May 2004

[5] Ying Yi, I. Nousias, M. Milward, S. Khawam, T. Arslan, I. Lindsay, "System-level Scheduling on Instruction Cell Based Reconfigurable Systems", *Proceedings of the Design Automation & Test in Europe Conference* , 2006

[6] Gu, Z. and Yuan, M. and He, X., "Optimal Static Task Scheduling on Reconfigurable Hardware Devices Using Model-Checking", *Real Time and Embedded Technology and Applications Symposium*, pages 32-44, 2007

[7] W. Fu, K. Compton, "An execution environment for reconfigurable computing", *IEEE Symposium on Field Programmable Custom Computing Machines*, April 2005.

[8] Heather Quinn, L.A. Smith King, Miriam Leeser, Waleed Meleis, "Runtime Assignment of Reconfigurable Hardware Components for Image Processing Pipelines,", *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003

[9] Y. Lu, T. Marconi, G. N. Gaydadjiev, K.L.M. Bertels, R. J. Meeuws, "A Self-adaptive on-line Task Placement Algorithm for Partially Reconfigurable Systems", *Proceedings of the 22nd Annual International Parallel & Distributed Processing Symposium* (IPDPS 2008) - RAW2008, pp. 8, Miami, Florida, USA, April 2008

[10] T. Marconi, Y. Lu, K.L.M. Bertels, G. N. Gaydadjiev, "Online Hardware Task Scheduling and Placement Algorithm on Partially Reconfigurable Devices", *Proceedings of International Workshop on Applied Reconfigurable Computing* (ARC), March 2008

[11] A. Ahmadinia, C. Bobda, S. P. Fekete, J. Teich, and J. C. van der Veen, "Optimal free-space management and routingconscious dynamic placement for reconfigurable devices", *IEEE Transactions on Computers*, volume 56, pages 673– 680, May 2007

[12] E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, "Compiling for the Molen Programming Paradigm", *Proceedings of the 13th International Conference on Field Programmable Logic and Applications* (FPL'03), pp. 900-910, Lisbon, Portugal, September 2003

[13] M. Sabeghi, K.L.M. Bertels, "Toward a Runtime System for Reconfigurable Computers: A Virtualization Approach", *Design, Automation and Tes int Europe*, Date09, 2009

[14] E. Moscu Panainte, K.L.M. Bertels, S. Vassiliadis, "The Molen Compiler for Reconfigurable Processors", *ACM Transactions in Embedded Computing Systems* (TECS), Volume 6 , Issue 1, February 2007

[15] T. Marconi, Y. Lu, K.L.M. Bertels, G. N. Gaydadjiev, "Intelligent Merging Online Task Placement Algorithm for Partial Reconfigurable Systems", *Proceedings of Design, Automation and Test in Europe* (DATE 08), March 2008

[16] S. Jovanovic, C. Tanougast, S. Weber, "A Hardware Preemptive Multitasking Mechanism Based on Scan-path Register Structure for FPGA-based Reconfigurable Systems,", *Second NASA/ESA Conference on Adaptive Hardware and Systems* , 2007

[17] V. Manh Tuan, H. and Amano, "A Preemption Algorithm for a Multitasking Environment on Dynamically Reconfigurable Processor", *In Proceedings of the 4th international Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, March 2008

[18] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov, E. Moscu Panainte, The Molen Polymorphic Processor, IEEE Transactions on Computers, pp. 1363- 1375, Volume 53, Issue 11, November 2004

[19] R. Lysecky, G. Stitt, F. Vahid, "Warp Processors", ACM Transactions on Design Automation of Electronic Systems (TODAES), Volume 11 , Issue 3, July 2006

[20] G. Barnett, CPU Scheduling Simulator, 2009, Online, Internet. Available WWW: http://cpuss.codeplex.com/