# Interfacing Operating Systems and Polymorphic Computing Platforms based on the MOLEN Programming Paradigm

Mojtaba Sabeghi, Koen Bertels

# Interfacing Operating Systems and Polymorphic Computing Platforms based on the MOLEN Programming Paradigm

**Mojtaba Sabeghi, Koen Bertels**
**Computer engineering Laboratory**
**Delft University of Technology**
**Delft, the Netherlands**
**{M.Sabeghi, K.L.M.Bertels}@tudelft.nl**

**Abstract – The MOLEN Programming Paradigm was proposed to offer a general function like execution of the computation intensive parts of the programs on the reconfigurable fabric of the polymorphic computing platforms. Within the MOLEN programming paradigm, the MOLEN SET and EXECUTE primitives are employed to map an arbitrary function on the reconfigurable hardware. However, these instructions in their current status are intended for single application execution scenario. In this paper, we extended the semantic of MOLEN SET and EXECUTE to have a more generalized approach and support multi application, multitasking scenarios. This way, the new SET and EXECUTES are APIs added to the operating system runtime. We use these APIs to abstract the concept of the task from its actual implementation. Our experiments show that the proposed approach has a negligible overhead over the overall applications execution.**

## 1 Introduction

Polymorphic computing platforms [1] usually consist of a General Purpose Processor (GPP) and reconfigurable unit(s) implemented in an FPGA technology. Programming such systems usually implies the introduction of a new software design flow which requires detailed knowledge about the reconfigurable hardware. The compiler is a very important component in the software design flow as it has to integrate most of this information.

To increase the system performance, computational intensive operations are usually implemented on the reconfigurable hardware. Different vendors provide their own implementation for each specific operation. The main challenge is to integrate these implementations - whenever possible - in new or existing applications. Such integration is only possible when application developers as well as hardware providers adopt a common programming paradigm.

The MOLEN programming paradigm [2] is a sequential consistency paradigm for programming reconfigurable machines. This paradigm allows parallel and concurrent hardware execution and it is currently intended for *single* program execution.

However, movement towards multi applications, multi tasking scenarios, adds new design factor to the system such as dealing with FPGA as a shared resource. These factors prevent using the MOLEN primitives as they are. They should be extended in such a way that besides offering the old functionalities, they have to resolves the conflicting issues between different applications at the time of primitive usage. In this paper, we present how the MOLEN programming paradigm primitives are extended and adapted into our runtime system.

The rest of the paper is organized as follows. Section 2 covers a summary over the related works. In Section 3, we present a background overview. Section 4 describes the runtime primitives followed by the evaluation results in section 5. Finally, we conclude the paper in section 6.

## 2 Related Work

The main challenge in general-purpose reconfigurable computers which serve multiple concurrent applications, is sharing the reconfigurable fabric in a transparent and light-weighted manner. Several research projects are intended to offer a consistent runtime system which can handle such a reconfiguration aware resource sharing. The IBM Lime [3] goal is to create a single unified programming language and environment that allows all portions of a system to move fluidly between hardware and software, dynamically and adaptively. Lime targets Java applications to be dynamically translated for co-execution on general-purpose processors and reconfigurable logic. Another similar work is PPL [4] which tries to extend the java virtual machine approach by featuring a parallel object language to be executed on a common parallel runtime system, mapping this language onto the respective computing nodes.

ReconOS [5] aims at the investigation and development of a programming and execution model for dynamically reconfigurable hardware devices. ReconOS extends the concept of multithreaded programming to reconfigurable logic. Another comparable approach is BORPH [6], which introduces the concept of hardware process that behaves just like a normal user program except that it is a hardware design running on a FPGA. Hardware processes behave like

* This research is partially supported by hArtes project EU-IST-035143, Artemisia iFEST project (grant 100203), Artemisia SMECY (grant 100230) and FP7 Reflect (grant 248976).

normal software programs. The BORPH kernel provides standard system services, such as file system access, to hardware processes, allowing them to communicate with the rest of the system easily and systematically.

Our work focuses on MOLEN programming paradigm and considers the FPGA as a co-processor rather than having complete hardware threads as in ReconOS and BORPH. From this Perspective, our work is more similar to the HybridOS [7] approach in which the granularity of the computation on the FPGA is based on multiple data parallel kernels mapped into accelerators to be accessed by multiple threads of execution in an interleaved and space-multiplexed fashion.

StarPU [8] offers a unified task abstraction named "codelet". Rather than rewriting the entire code, programmers can encapsulate existing functions within codelets. StarPU takes care to schedule and execute those codelets as efficiently as possible over the entire machine. In order to relieve programmers from the burden of explicit data transfers, a high-level data management library enforces memory coherency over the machine: before a codelet starts, all its data are transparently made available on the computing resource.

We are targeting the tightly coupled processor coprocessor MOLEN paradigm in which we abstract the concept of the task using MOLEN SET and EXECUTE instructions.

# 3 Background Overview

The MOLEN hardware organization 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. **Figure 1** represents the MOLEN machine organization.

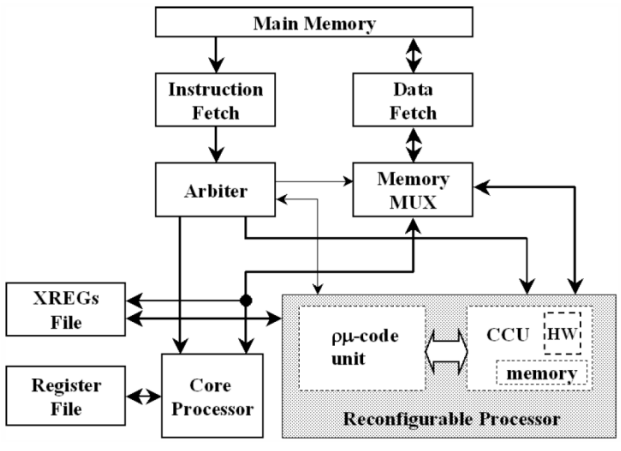


**Figure 1 MOLEN Hardware Organization**

## 3.1 MOLEN Programming Paradigm

MOLEN programming paradigm presents a programming model for reconfigurable computing that allows modularity, general *function like* code execution and parallelism in a sequential consistency computational model. Furthermore, it defines a minimal ISA extension to support the programming paradigm. Such an extension allows the mapping of an arbitrary function on the reconfigurable hardware with no additional instruction requirements.

This is done by introducing new *super instructions* to operate the FPGA from the software. An operation, 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. These phasing introduces two super instructions; SET and EXECUTE.

The SET instruction requires single parameter – e.g. the beginning address of the configuration microcode. When a SET instruction is detected, the Arbiter reads every sequential memory address until the termination condition is met and configures it on the FPGA. After completion of the SET phase, the hardware is ready to be used for the targeted functionality. This is done using the EXECUTE instruction. This instruction also utilizes a single parameter being the address of the execution microcode. The execution microcode performs the real operation which consists of reading the input parameters, performing the targeted computation and writing the results to the output registers.

As it is obvious, these two instructions are based on the assumption of a single thread of execution. With such an assumption, having an operating system as long as there is only one application dealing with the FPGA is not an issue. That is because there is no competition for the resources and the application has full control over the FPGA. In case of serving several concurrent applications on the same system, SET and EXECUTE can not be used the same way as they are used in single application paradigm. Each application might issue its own SET (EXECUTE) which most probably has conflicts with the other's SETs (EXECUTEs). In such a scenario, the operating system has to resolve all the conflicts.

In the next section, we describe our runtime execution environment in which the MOLEN primitives are used to operate the FPGA.

## 3.2 The Runtime Environment

Our runtime environment [9] is 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.

The runtime environment components include a scheduler, a profiler and a transformer. It might also incorporate a JIT compiler for on the fly code generation for the target cores,

e.g. FPGA bit streams. Figure 2 depicts our runtime environment.

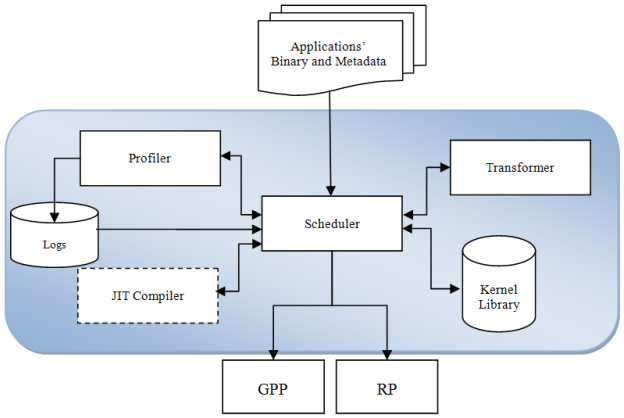


**Figure 2 the Runtime Environment**

In our system, task scheduling takes place in two phases. First, at compile-time, the compiler performs static scheduling of the reconfiguration requests (SETs and EXECUTEs) assuming a single application execution. The main goal at this stage is to hide the reconfiguration delay by configuring the operations well in advance before the execution point.

Then at runtime, the run-time system performs the actual task scheduling. At this stage, the MOLEN SET and EXECUTE instructions are just a hint and they do not impose anything to the runtime system. The run-time system decides based on the runtime status of the system and it is possible to run a kernel in software even though the compiler already scheduled the configuration. More detail about the scheduling procedure can be found in [10]. In this paper, we only focus on the runtime SET and EXECUTE operations.

We also have a kernel library which includes a set of precompiled implementation for each known operations. This means, we might have multiple implementations per operation. Each implementation has different characteristics which are 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.
For each operation's implementation in the library, there is a software wrapper which is kept in the form of a Dynamic Shared Object (DSO). The application developer can also provide his own DSO along with the required metadata. To this end, we provide the application developers with a DSO creation tool, which is discussed later.

## 4 MOLEN Runtime Primitives

To keep the changes in the compiler and design tool chain [11] as limited as possible and also to provide legacy compatibility, we propose the MOLEN runtime primitives as follows.

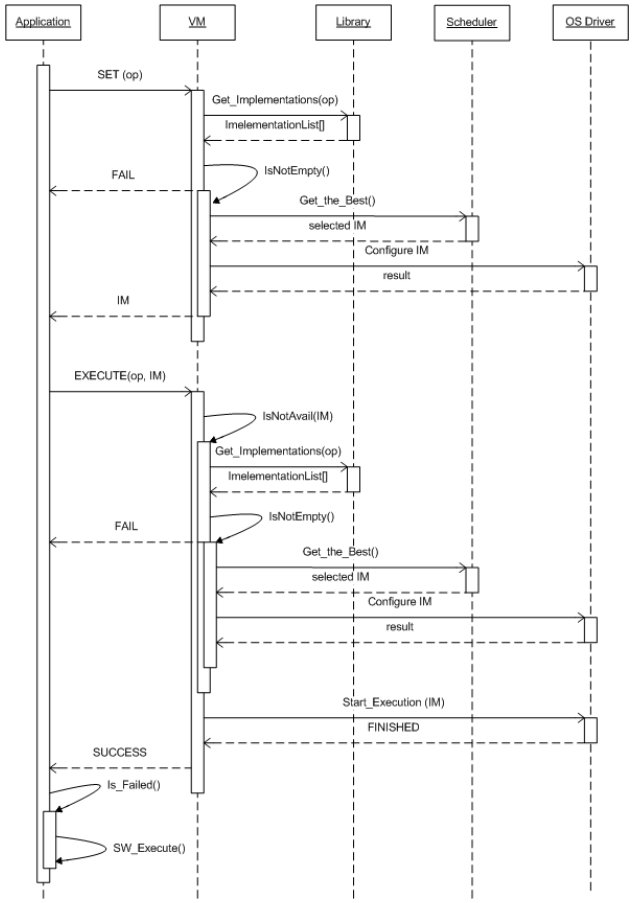


**Figure 3 the Operation Execution Process**

We have extended the operating system runtime with two APIs; The MOLEN SET and MOLEN EXECUTE. The functionality of these APIs are almost identical to the original MOLEN SET and EXECUTE. Besides the normal MOLEN activities, these APIs have to take care of the sharing of the FPGA among all the competing applications. This means, at the time of the call, the runtime system is responsible to check the availability of the FPGA. Furthermore, it can impose some sort of allocation policies such as priorities and performance issues.

Figure 3 shows the sequence diagram of the operation execution in our runtime system. When an application comes upon a call to the SET for a specific operation, it sends its request to the runtime system (VM). The VM then checks the library to look for all the appropriate implementations. If no such implementation is found, it sends a FAIL message back to the application which means the SET operation can not be performed. Otherwise, based on the scheduling policy it selects one of the implementations (IM) and configure it on the FPGA. The OS driver is the low level interface between the operating system and the physical FPGA fabric. Finally, it sends the address of the IM to the application.

Later on, when the application encounters the EXECUTE instruction, the system checks if the IM is still configured and ready. If so, the execution can start right away. If not, it

has to follow the SET routine again and at the end, starts the execution. If any problem occurs during this process, a FAIL message will be sent back to the application. A FAIL message received by the application means the software execution of the operation has to be started. In the following two sections, we describe the two APIs in more detail.

### 4.1 MOLEN SET

The SET API receives the operation name as an input. We assume all the supported operations have a unique name in our system. This assumption is based on the idea of having a library of a number of different implementations per operation in our runtime environment. Listing 1 shows the pseudo code corresponding to the SET API.

**Listing 1** the SET

```
SET (input: Operation op): return
Implementation I*
1- SET begins
2- Assume im_list the list of all the
   implementations corresponding to the
   op in the library;
3- Assume co_list as an empty list;
4- For each IM in im_list
   4-1- If the corresponding physical
   location of IM is busy
       Remove IM from im_list;
       Continue;
   End if
   4-2- If IM is already configured on
   the FPGA, Add IM to the co_list;
End for
5- If co_list is not empty
   Return the IM with the minimum
   execution time from co_list;
6- If im_list is not empty
   6-1- Choose I* from the im_list based
   on the scheduling policy;
   6-2- Configure I* on FPGA;
   6-3- Return I*;
End if
7- Return FAIL;
8- SET ends
```

In Listing 1, line 2 creates a list of all the existing implementation for the operation. If the physical location corresponding to any of those implementations is busy, e.g. another application is using that resource, that implementation is removed from the list in line 4-1 and the loop continues to the next element in the list. Some of the implementations might already be configured on the FPGA. This means there is no need for configuring them again. Those implementations are added to another list in line 4-2 and the best candidate (here the fastest one) is return to the main program in line 5. If there is no such an implementation exists, the algorithm goes further to choose one of the other implementations and start configuring it in line 6. This selection is very dependent on the scheduling (line 6-1). The configuration process is discussed in section 4.3.

### 4.2 MOLEN EXECUTE

The EXECUTE is also an API added to the operating system. It has two input arguments; the operation name and the address of the configured implementation in the SET phase. Listing 2 shows the pseudo code corresponding to the EXECUTE.

In our system, the operations might be shared between different applications (This task sharing is one of the motivations behind the idea of using dynamic shared object as will be discussed in section 4.3). On the other hand, since there might be a gap between the occurrence of the SET and EXECUTE instructions, e.g. because of the compiler optimizations to hide the reconfiguration delay, the control might go to another application (app2) and that application (app2) might use the implementation which is set by this application. That is why the busy status of the *IM (*in line 2) has to be checked. If it is not busy, it can start execution. It is also possible to call the EXECUTE without any prior SET or any successful prior SET.

**Listing 2** the EXECUTE

```
EXECUTE (input: Operation op; input:
Implementation IM)
1- EXECUTE begins
2- If IM is not NULL and IM is not busy
       Execute IM;
       Return SUCCESS;
End if
3- I* = SET (op);
4- If I* is not NULL and I* is not busy
       Execute I*;
       Return SUCCESS;
End if
5- Return FAIL;
6- EXECUTE ends
```

In this case *IM* is null. In case of having a busy implementation or a null, the SET has to be performed again. This is done in line 3. Finally, the algorithm executes the implementation in line 4. If any problem occurs during the EXECUTE, it return a FAIL which means the operation has to be executed in software. The execution process is discussed in section 4.3.

### 4.3 Dynamic Binding Implementation

As we pointed earlier, the actual binding of the function calls to the implementation happens at runtime. To do that we use

the ELF binary format delayed symbol resolution facility and position independent code.

For each operation implementation in the library, there is a software wrapper with two functions, one which performs the low level configuration of the operation (the traditional SET) and one which performs the low level execution of the operation (the traditional EXECUTE). In the runtime SET, when the reconfiguration takes place (line 6-2 in Listing 1), the low level SET from this software wrapper is called. Similarly, in the runtime EXECUTE (lines 2 and 4 in Listing 2) the low level EXECUTE is called. The reason that we can use the traditional SET and execute at this point is that the sharing controls has already been performed by the runtime system and it is safe to call the normal SET and EXECUTE instruction.

As it is mentioned before, this software wrapper is kept in the form of a Dynamic Shared Object (DSO). Given the name of a DSO by the SET (line 6-2 in Listing 1), which is the name of the chosen implementation; the system dynamically loads the object file into the address space of the program and returns a handle to it for future operations. We do this process by using the Linux *dlopen* function. The *dlopen* is called with RTLD_LAZY mode, which says to perform resolutions only when they're needed. This is done internally by redirecting all requests that are yet to be resolved through the dynamic linker. In this way, the dynamic linker knows at request time when a new reference is occurring, and resolution occurs normally. Subsequent calls do not require a repeat of the resolution. To find the address of each function in the DSO, we use Linux *dlsym* facility. The *dlsym* takes the name of the function and returns a pointer containing the resolved address of the function.

In the traditional SET (line 6-2 in Listing 1), all the required parameters needed by the FPGA have to be transferred to MOLEN XREGS. Then, it starts configuring the FPGA. At the time of traditional EXECUTE (lines 2 and 4 in Listing 2); using the *dlsym* the address of the second function is resolved. By this function pointer, we can invoke the required operation.

To simplify the creation of DSO files to be added to the runtime library, (especially for third-party modules) a support tool is proposed. The idea is simple: It shows a template of the wrapper and the program developer has to add a few lines of code to it. Besides, the program developer has to explicitly write the parameters transfers instruction in the pre defined template (moving the parameters to XREGs). Then, the tool compiles the code for Position Independent Code (PIC) and converts it to a DSO. Furthermore, the tool provides a very simple interface to gather the metadata required by the runtime scheduler such as the configuration latency, execution time, memory bandwidth requirements, power consumption, physical location on the reconfigurable fabric, etc and stores them in an appropriate format.

# 5 Evaluation

When evaluating our proposed mechanism, two aspects are important: what is the overall performance improvement through acceleration which can be achieved and what the overhead of invoking it is.

**Overhead:** The execution time overhead imposed by dynamic linking (DSO loading) occurs on two places: at run and load-time. At runtime, each reference to an externally defined symbol must be indirected through the Global Object Table (GOT). The GOT contains the absolute addresses of all the static data referenced in the program. At load-time, the running program must copy the loaded code and then link it to the program. In most cases, the only run-time overhead of dynamic code is the need to access imported symbols through the GOT. Each access requires only one additional instruction. The load time overhead is the time spent to load the object file. For a null function call in our system, the load time is about 0.75 milliseconds. For a typical wrapper function, the load time increases to about 2 milliseconds. We should mention that the increase in the input parameters' size might increase the size of the wrapper function since each parameter needs a separate instruction to be transferred to the MOLEN XREGs.

**Speedup**: In order to show the overall performance of the system, we performed a series of experiments. To show only the overhead imposed by the SET and EXECUTE APIs, we have implemented a scheduling algorithm in which we pick the fastest implementation and execute it, on condition of course that the FPGA is available. The experiment workload is obtained from an interactive multimedia internet based testing application [12]. The workload's kernels are listed in Table 1.

The last column in Table 1, shows the operation total execution time when it is executed only once. This means the execution time is the sum of the software wrapper load delay plus the reconfiguration delay plus the HW execution time. As shown in Table 1, the software wrapper delay over the total execution time varies between 5 to 20 percent for different kernels.

However in general, when a kernel is loaded (incurring one wrapper and reconfiguration delay), it executes more than once which means the overhead decreases as the number of executions increases. To show such a reduction in execution time, we evaluate the overall execution time in the following. The first column is the software only execution time (no FPGA) which is mentioned just as a point of reference.

To show overall system performance, we used 5 different workloads from interactive multimedia internet based test; the workload varies based on the number of test taken and the number of kernels which is used in each test. We have workloads for 12 applicants (821 kernels), 24 applicants (1534 kernels), 36 applicants (2586 kernels), 48 applicants (3032 kernels) and 60 applicants (4164 kernels). It should be

mentioned that each test taker is has its own process in the system and therefore the number of applications are equal to the number f test takers. In such a scenario, each test takers corresponding application is competing against the others to obtain the FPGA resources.

We compared the software only execution with the hardware/software execution. As shown in Table 2 the overall system speedup varies between 2.28 to 1.91. The wrapper overhead to the overall execution time is between five to three percent. As the number of test takers increases, the chance of executing an already configured kernel increases and as a result, the wrapper overhead reduces.

On the other hand, since the system loads increases, the overall speedup is also decreases. That is because the FPGA resources are limited and fixed. Therefore, when the system load increases the HW/SW execution time gets closer to the SW only solution and as a result the speedup reduces.

**Table 1 Workload Kernels**

| Kernels | Operation SW only execution time (*ms*) | Operation HW Execution time (*ms*) | Operation Configuration Delay (*ms*) | Operation SW wrapper Delay (*ms*) | The HW total execution time (ms) |
|---|---|---|---|---|---|
| Epic-Decoder | 19.87 | 8.56 | 5.82 | 2.11 | 16.49 |
| Epic-Encoder | 11.87 | 5.22 | 2.49 | 1.17 | 8.88 |
| Mpeg2-Decoder | 77.35 | 2.43 | 3.64 | 1.47 | 7.54 |
| Mpeg2-Encoder | 10.39 | 1.94 | 4.87 | 1.81 | 8.62 |
| G721 | 42.42 | 4.64 | 5.82 | 2.57 | 12.58 |
| Jpeg-Decoder | 68.39 | 8.63 | 8.72 | 3.41 | 20.76 |
| Jpeg-Encoder | 169.33 | 35.23 | 10.98 | 4.51 | 50.72 |
| Pegwit | 166.06 | 36.34 | 5.88 | 2.59 | 44.81 |

**Table 2 Overall Execution Time**

| No application No kernels | 12 821 | 24 1534 | 36 2586 | 48 3032 | 60 4164 |
|---|---|---|---|---|---|
| SW-only | 135654.08 | 260508.60 | 381329.44 | 501860.74 | 641478.23 |
| SW/HW | 59580.79 | 121977.13 | 186415.10 | 256929.84 | 335276.90 |
| SW wrapper Overhead | 2983.03 | 5884.87 | 7654.71 | 10814.62 | 11463.15 |
| Wrapper overhead percentage | ~ 5 % | ~ 5 % | ~ 4 % | ~ 4 % | ~ 3 % |
| Speedup | 2.28 | 2.14 | 2.05 | 1.95 | 1.91 |

# 6 Conclusion

In this paper, we extended the MOLEN programming paradigms primitives to use them in presence of an operating system and in multi application, multi tasking scenarios. The MOLEN primitives in their current status are just for single application execution. We discussed the details of the SET and EXECUTE APIs and presented the dynamic binding mechanism whish is used by these APIs to bind a task call to a proper task implementation. Our experiments show that the proposed approach has a negligible overhead over the overall applications execution.

# References


1. Vassiliadis, S., Kuzmanov, G.K., Wong, S., Panainte, E.M., Gaydadjiev, G.N., Bertels, K.L.M., Cheresiz, D.: PISC: Polymorphic Instruction Set Computers. International Workshop on Applied Reconfigurable Computing (2006)

2. Vassiliadis, S., Gaydadjiev, G.N., Bertels, K.L.M., Panainte, E.M.: The Molen Programming Paradigm. Third International Workshop on Systems, Architectures, Modeling, and Simulation (2003)

3. Huang, S.S., Hormati, A., Bacon, D.F., Rabbah, R.: Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. European Conference on Object-Oriented Programming (ECOOP) (2008)

4. Olukotun, K.: Towards Pervasive Parallelism. (2010)

5. Lübbers, E., Platzner, M.: ReconOS: An Operating System for Dynamically Reconfigurable Hardware. Dynamically Reconfigurable Systems. Springer (2010)

6. So, H.K.-H., Brodersen, R.: A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. ACM Transactions on Embedded Computing Systems (TECS) **7** (2008)

7. Kelm, J.H., Lumetta, S.S.: HybridOS: Runtime Support for Reconfigurable Accelerators. International Symposium on Field-Programmable Gate Arrays, Monterey, California (2008)

8. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. 15th International Euro-Par Conference (2009)

9. Sabeghi, M., Bertels, K.: Toward a Runtime System for Reconfigurable Computers: A Virtualization Approach. Proceedings of the conference on Design, automation and test in Europe (2009)

10. Sabeghi, M., Sima, V.M., Bertels, K.L.M.: Compiler Assisted Runtime Task Scheduling on a Reconfigurable Computer. 19th International Conference on Field Programmable Logic and Applications (FPL09) (2009)

11. Panainte, E.M., Bertels, K., Vassiliadis, S.: Compiling for the Molen Programming Paradigm. Field-Programmable Logic and Applications (2003) 900-910

12. Fazlali, M., Zakerolhosseini, A.: REC-BENCH: A Tool to Create Benchmark for Reconfigurable Computers. VI Southern Programmable Logic Conference, (SPL 2010) (2010)