Toward a Runtime System for Reconfigurable Computers: A Virtualization Approach

Mojtaba Sabeghi, Koen Bertels
Computer Engineering Laboratory
Delft University of Technology
Delft, the Netherlands
{sabeghi, koen}@ce.et.tudelft.nl

***Abstract*—In this paper we propose a virtualization layer to handle the program execution on reconfigurable computers in order to address one of their biggest problems which is the management of the reconfigurable hardware in a multi-tasking environment. The virtualization layer is responsible for allocating the hardware at run-time based on the status of the system. Furthermore, it provides a consistent and low overhead interface to decouple the process of software development from hardware design which will result in the software to be independent of the underlying reconfigurable hardware. This paper discusses the virtual layer's specification and components. Our preliminary results for a prototype simulated on Molen hardware organization show a competitive performance comparing with an optimal hardware allocation.**

Keywords-component; run-time support, reconfigurable computers, virtualization

I. INTRODUCTION

In recent years, reconfigurable architectures have received ever increasing attention due to their adaptability and short design time. The main advantage of reconfigurable computing is its ability to increase the performance with accelerated hardware execution, while possessing the flexibility of a software solution. Reconfigurable systems can speed up the application's execution time by mapping selected application parts, called kernels, onto reconfigurable hardware.

However, current efforts [1] for designing reconfigurable systems are not flexible as they manage the hardware with static and design time decisions based on fixed rules. Most of them have assumed only a single thread of execution, where the given application has full ownership of both the host microprocessor and the reconfigurable logic. But, considering multiple applications on a system which are executing concurrently and competing for the resources, these static mapping cannot be of much help. There should be a runtime system that can decide based on the runtime conditions which parts of the applications should be executed on the hardware. This runtime system is supposed to efficiently operate the system and resolve the conflicts between the executing tasks.

On the other hand, the application developer cannot target reconfigurable computers easily as it requires a substantial knowledge of hardware design. One of the major trends toward solving this problem is to develop tool chains that can automatically detect the parts of the program that can be accelerated in hardware and perform all the processes automatically. However, these methods are still static and design time approaches.

The Warp Processors project [2] is intended to address the runtime allocation and automatic hardware generation from a normal binary. However, they only target single application paradigm. Compton et al in [3] proposes that an application be distributed with both software and hardware descriptions of compute-intensive kernels. And, the scheduler decides which one to be executed at runtime. However, the portability of the applications across different hardware platforms is missing here.

Vforce [6] is designed to allow the same application code to run on different reconfigurable computing platforms, and to permit the runtime binding of applications to hardware. In this approach, the application developers must be aware of the kernels and call a generic hardware object for each kernel. The runtime system then is responsible to provide the exact implementation for the specific platform. In our opinion, this is just a kind of binding and the real decision about hardware or software execution has been taken on design time.

To solve these problems, we propose a consistent, protected, low overhead interface which decides how to allocate the hardware at run-time based on the status of the system. Moreover, this layer will hide all platform dependent details and provides a transparent application development process. This layer can take place above the Operating System.

978-3-9810801-5-5/DATE09 © 2009 EDAA

In this paper, we use the Molen hardware platform [4] and Molen programming paradigm [5] for our discussion. However, the same approach can be applied on the other hardware platforms.

Molen is established on the basis of 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. 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 operation and in the execute phase the actual execution of the operation 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 as it is depicted in figure 1 consists of two parts; the general purpose processor (GPP) and the reconfigurable processor (RP) usually implemented on an FPGA. Another key component is the Arbiter which performs a partial decoding of the instructions received from the instruction fetch unit and issues them to the appropriate processor (GPP or RP). The Exchange Registers (XREGs) are used for data communication between the Core Processor and Reconfigurable Processor. Parameters are moved from the register file to the XREGs and the results stored back from the XREGs in the register file.

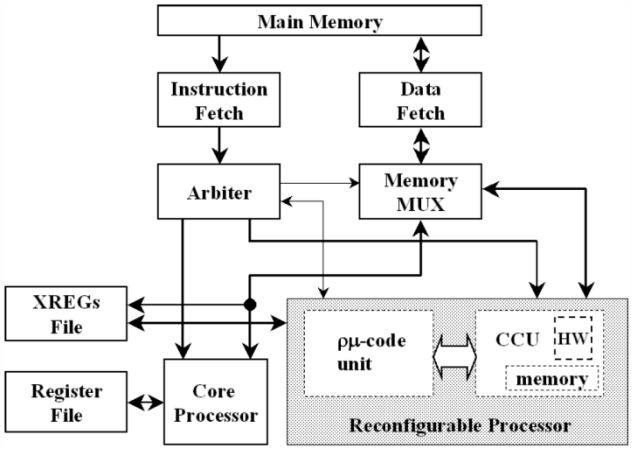

Fig. 1. Molen Hardware Organization

The rest of the paper is organized as follows. Section 2 describes the execution model in the proposed virtualization layer. Section 3 presents the components of the virtualization layer. Section 4 talks about the preliminary results follows by section 5 which conclude the paper.

II. THE PROPOSED MODEL

The execution can take place either on the general purpose processor or on the reconfigurable processor based on the runtime dynamic condition. This decision is very dependent on the system and the metrics that need to be optimized. It can be performance, power consumption or a smaller footprint.

To be able to run on the reconfigurable hardware, the virtualization layer needs to inspect, analyze and do binary modification to the applications. The layer monitors the programs binary to find the computation intensive parts, the kernels, and if they can run on hardware, replace them with the call to the hardware implementation.

Once those replacements have been applied, the extracted kernels have to be implemented in hardware. This can be done either automatically with the help of a JIT compiler which converts the binary to the bitstream or using an available implementation from a library. A JIT Compiler generates considerable overhead and yet, there is no fast and reliable JIT compiler to translate the software binary to the reconfigurable hardware implementation. Therefore, having a library of the common kernels implemented for each hardware platform is a very good alternative. Besides, during the software installation process, new kernels which are needed by the application can be added to the library.

One of the problems with the library is the matching between the selected part in the application and the equivalent kernels in the library which offers the same functionality. As there is no good solution for functionality matching so far, we propose an alternative solution by assigning each kernel a unique identifier. The matching between the kernel in the application and the library can be based on this identifier.

However, this might impose an extra work on the programmers as they need to annotate their source code with these identifiers. To simplify this, we can also have a software class library equivalent to the hardware kernel library. Each kernel in the hardware library has a software implementation in the class library and the programmers can use this class library during the software development.

III. THE LAYER COMPONENTS

Here, we describe the components of the virtualization layer depicted in figure 2:

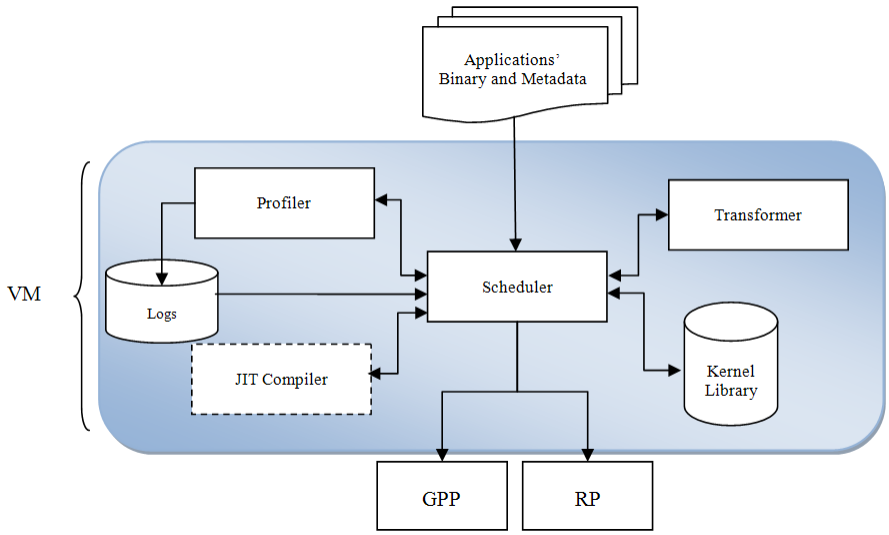

Figure 2. The Virtualization Layer Components

A. *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 but it can also be any cluster of instructions that is scattered throughout the application.

During the program execution, the scheduler monitors the applications binary and intercepts the kernels. Then, it estimates the potential speedup that can be achieved when the kernels are executed on the reconfigurable hardware and estimates the initial cost of a hardware mapping. After that, based on the scheduling policy and considering other applications requirements, the scheduler decides how to allocate the hardware. Any scheduling policy can be used here for example the Most Frequently Used, the Best Speedup and the Multi Constraint Knapsack presented in [3].

Intercepting the kernels in run time is not a trivial task. It requires complex runtime profiling tools which might impose considerable overhead. Using design time information can be a viable alternative. The compiler at compile time can instrument the binary based on those information. Within the Molen programming paradigm [5], the set and execute instructions can be effectively used as a mean for instrumentation. However, these instructions do not configure or execute anything on the hardware although they are meant to do so. The set just informs the scheduler of a possible future call to a hardware and the execute instruction is a signal to the system to execute the hardware if that is possible. Both set and execute instructions includes the corresponding kernel identifier. Whenever the system encounters a set or an execute instruction, it invokes the scheduler to decide.

The scheduler can also utilize more complex scheduling policies. Those policies can be based on the information from the design time, information collected at runtime or even some heuristics. A good source of information from design time is the Configuration call graph (CCG) which shows the future of the system and can help in finding a near optimal schedule. The CCG is a directed graph presenting the kernels identified by the profiler. Each node in this graph contains the kernel identifier which uniquely identifies the kernel. The edges of the graph represent the dependencies between the configurations within the application.

B. *The Profiler*

The Profiler continually tracks the application behavior and records statistics such as the number of references to one kernel. These statistics when combined with the scheduling policy, are used to determine where, when and how to execute the kernels (hardware or software). In addition, the profiler can be used to find the kernels. The profiler must log the collected data in very fast and efficient data structures because retrieving them should not put any burden to the system performance.

C. *The Transformer*

The transformer replaces the software implementation of the kernel with a call to the hardware. In fact, it uses a binary rewriting mechanism and can again impose considerable overhead. This mechanism has to assure the correct input/output parameters transfer. However, within the Molen programming paradigm, we propose a mechanism which does not require binary rewriting. The

compiler can put the set and execute instructions besides the software version of the kernel in a conditional structure. The condition can be based on an environment variable. The virtualization layer sets that variable based on the scheduling decision.

D. Kernel Library

The kernel library is a precompiled and already synthesized set of kernels implemented for the underlying reconfigurable hardware. For each kernel, there might be a couple of different versions (with the same identifier) in the library each of which differs with the others in one or more metrics for example logic size or/and power consumption. Anyway, it does not impose any trouble to the application developer as it is completely transparent to them.

Corresponding to each version, the library includes some metadata describing the kernel's characteristics such as the configuration latency, execution time, memory bandwidth requirement, power consumption and logic size. These metadata are mainly being used by the scheduler based on the scheduling policy. Furthermore, it contains the physical mapping location of the kernel on the FPGA.

This library can be synthesized for each reconfigurable hardware platform resulting in the applications, independent of the underlying platform.

E. JIT Compiler

This is a just-in-time compiler that can be used to compile the kernels for which there is no implementation in the library. The compiler converts binary to bitstream. Any just in time compiler can be used here nevertheless there is no efficient one yet available.

IV. PRELIMINARY RESULTS

We made a simulation of the system with a very simple scheduling policy and based on the Molen programming paradigm. We have initiated several applications with a couple of kernels running concurrently competing for the reconfigurable hardware.

The scheduling algorithm compares the software execution time of a kernel with the hardware execution time plus the hardware configuration latency. If the latter is smaller, the kernel is a candidate for hardware execution. If the reconfigurable hardware is available and there is no conflict, the kernel will be mapped to the hardware and executed. If there is a conflict, the scheduler looks at the kernel frequency of use obtained from the profiler log and selects the kernel with higher frequency.

As it is shown in figure 3, the system has a competitive performance compared to the optimal hardware allocation. The optimal hardware allocation has been done manually and is based on the allocation of the best possible kernels on the hardware by looking further ahead than the current execution point.

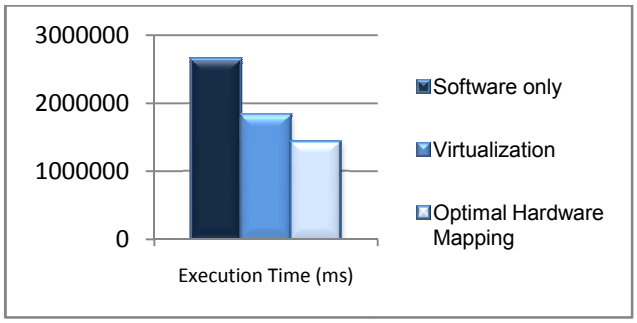

Figure 3. Execution time for the software only execution, virtualized execution and Optimal hardware mapping execution

V. CONCLUSION AND FUTURE WORKS

In this paper we presented a virtualization layer as a runtime system for reconfigurable computers. We briefly described the layer's component and the execution policy. In the future, we will work on a detailed specification of each component. Furthermore, we have to verify the system with extensive experiments and comparison with other runtime systems.

REFERENCES

[1] Katherine Compton, Scott Hauck, Reconfigurable computing: a survey of systems and software, *ACM Computing Surveys,* vol. 34, no. 2, pp 171-210, June, 2002

[2] Roman Lysecky, Greg Stitt, Frank Vahid, Wrap Processors, *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 659-681, July 2006.

[3] Wenyin Fu, Katherine Compton, An Execution Environment for Reconfigurable Computing, *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines,* pp. 149-158, April, 2005

[4] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, Elena Moscu Panainte, The MOLEN Polymorphic Processor, *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363-1375, 2004.

[5] Stamatis Vassiliadis, Georgi Gaydadjiev, Koen Bertels, Elena Moscu Panainte, The Molen Programming Paradigm, *Third International Workshop on Systems, Architectures, Modeling, and Simulation*, pp. 1-10, July 2003

[6] Nicholas Moore, Albert Conti, Miriam Leeser, Writing Portable Applications that Dynamically Bind at Run Time to Reconfigurable Hardware, *International Symposium on Field-Programmable Custom Computing Machines*, pp. 229-238, 2007