IP Based Design 2002

Session: 3A

IP Design for Dynamically Reconfigurable SoCs

Tobias Oppold, Wolfgang Rosenstiel
University of Tuebingen
Wilhelm-Schickard-Institute
Sand 14, 72076 Tuebingen, Germany
{oppold,rosenstiel}@informatik.uni-tuebingen.de

Abstract

Reconfigurable computing is a relatively new paradigm. While there is a growing number of new reconfigurable architectures there is a lack of design tools to map applications onto these architectures. The design tools must exploit the benefits of reconfiguration in order to deploy such architectures successfully. In this paper we present some fundamental ideas on how to take advantage of reconfiguration and how to automate the design of IP that is running on architectures that can be reconfigured very fast.

1 Introduction

Designers of digital systems have to meet many different requirements. Especially for mobile applications, low energy consumption can be a crucial factor. Very often, high performance is also an important issue. If the system is flexible, changing standards can be implemented fast and a shorter time to market can be achieved. And of course the costs need to be considered. System designers have to consider all these factors to implement a system successfully and often have to make trade-offs between the conflictive requirements. A fundamental decision is whether some functionality will be implemented in software or in hardware. Both approaches have merits and drawbacks. Reconfigurable hardware tries to combine the assets of both.

In the following section we will point out the different characteristics of software and hardware and how reconfigurable hardware can be classified. Section three will present some general considerations on how reconfiguration can be profitable and the subsequent section will outline a concrete approach to map applications onto dynamically reconfigurable hardware.

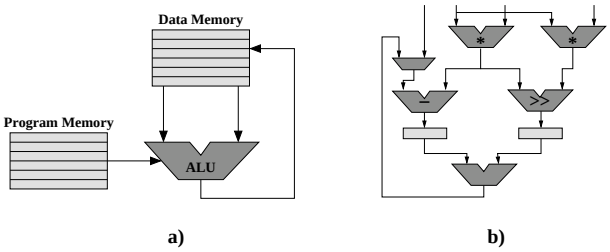

Figure 1. Characteristics of a) software and b) hardware

2 Software or hardware

Both approaches have their own characteristics. A software solution is characterized by the temporal processing of the application: the program is stored in a memory and is executed sequentially. Likewise, the application data is stored in a memory and can be accessed only sequentially (figure 1a). Due to the re-programmable nature of the RAM, software systems are very flexible allowing for a short time to market.

In contrast to that a hardware implementation is characterized by the spatial mapping of an application: the operators and operands are distributed in space along a custom data path (figure 1b). Due to the custom data path that is highly optimized for a given application, this approach offers high performance and low energy consumption.

The difference between these two paradigms is especially strong if we consider a general purpose standard microprocessor and a hardwired circuit. The standard processor is very flexible but with respect to performance and energy consumption it will not be optimal for a given application. In contrast to that a hardwired circuit would be very fast and energy efficient but not flexible at all. Besides these two extremes there are solutions that try to utilize the merits of the other approach.

The performance and energy consumption of a standard

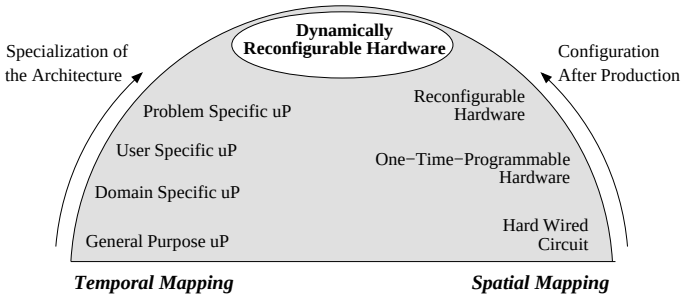

Figure 2. Dynamically reconfigurable hardware combines properties of both the software and the hardware approach

processor can be optimized by increasing the spatial component. This can be done reasonably by specializing the processor architecture for a certain application domain like digital signal processing or networking. The specialization can be further improved if a designer can specify their application domain individually. In this case the architecture supplies exactly the execution units needed for a certain application. This leads to a more efficient utilization of the deployed resources and therefore an improvement of performance and energy consumption can be expected. An example for such a configurable processor is Tensilica's Xtensa Architecture [10]. The spatial component can be emphasized even more if the architecture is not only optimized for a certain problem domain but for a concrete problem. Adelante [1] is doing this for instance by synthesizing a VLIW processor from a high level description.

A similar development can be observed on the hardware side where a temporal component is added to the traditional hardwired solution. This temporal component is realized by programmability of the device after fabrication. Programmable hardware like antifuse-FPGAs can speed up the time to market significantly. Reconfigurable hardware like SRAM-based FPGAs is even more flexible. They can be reconfigured many times even after shipping of the product. In [4] the characteristics of such reconfigurable computing architectures are defined as a) they can be customized to solve *any* problem after device fabrication and b) they exploit a large degree of spatially customized computation in order to perform their computation.

This definition says nothing about the time needed for customizing the device. The time needed to reconfigure an FPGA is in the range of microseconds to milliseconds. We use the term *dynamically reconfigurable hardware* (DRHW) to denote reconfigurable computing architectures (as defined above) that can be reconfigured within the range of one clock cycle. Due to the property of fast reconfiguration DRHW provides the capability to map an application both spatially and temporally. Figure 2 shows that DRHW can not be classified as either software or hardware but as combination of both approaches. We therefore use the more general term IP to denote an application, or a piece thereof, running on DRHW.

Architectures that match the definition of DRHW are for example NEC's Dynamically Reconfigurable Logic Engine (DRLE) [5], PACT's eXtreme Processor Platform (XPP) [8], and Chameleon Systems' Reconfigurable Communications Processor (RCP) [3]. An overview of other commercial reconfigurable platforms can be found in [9], academic reconfigurable platforms are surveyed, e. g., in [6].

3 General considerations

There are certainly applications that need an ASIC or even a full custom VLSI to satisfy the application's performance or power requirements. However, time to market, flexibility, and NRE-costs are very important factors and therefore FPGAs are gaining more and more market shares. While updating a system by reconfiguration of FPGAs is supported by commercial applications [11] loading different configurations onto FPGAs in order to have different functionalities over time is mainly in the domain of academic research. The fine granularity of FPGAs implies a high amount of reconfiguration data that need to be stored and transferred for reconfiguration. This makes frequent reconfiguration cycles inefficient and leaves only little space for suitable applications.

Dynamically reconfigurable devices tend to be more coarse grained [6] so that the relatively small amount of reconfiguration data can be stored on-chip. Storing multiple configurations on-chip removes the bottleneck to the external memory needed for traditional SRAM-FPGAs. Our definition of DRHW therefore requires a SoC comprising at least the reconfigurable fabric and the configuration memory. Other parts of a system might be on the same chip as well so that the DRHW might be only one part of a *dynamically reconfigurable SoC*.

In most applications not all parts of a circuit need to be available at a given instant of time. For example there is no need for a digital camera to compress and decompress a picture at the same time. If one decides to implement both algorithms in hardware large parts of the circuit will be idle at any given time. It is therefore obvious to deploy a reconfigurable architecture, hold the configuration data in memory, and to load the configuration data onto the reconfigurable fabric as it is needed. This can be profitable in terms of energy consumption, production costs, and even performance. There are many factors like, e. g., the above mentioned different granularity of FPGAs and DRHW that influence those properties. In order to be independent of such factors we want to point out the merits of reconfiguration under the assumption that the same reconfigurable fabric is used with or without reconfiguration:

Energy consumption - As a rule of thumb the energy consumption increases with the number of transistors needed to implement the circuit. Reconfiguration helps to reduce the number of transistors by sharing the reconfigurable fabric if the overhead for reconfiguration including the memory needed to store the configurations is low.

Production costs - The number of transistors is also connected with the costs for the production of a device. The costs for the final product that uses a circuit increases in particular if multiple devices must be used because process technology does not provide enough transistors on a single chip.

Performance - An important prerequisite for successful dynamic reconfiguration is the ability to share application data between different configurations. This can be considered a third dimension for routing. If an application is inherently divided into parts that do not need to run at the same time but need to access the same data, routing the data to both parts can have a strong impact on the overall clock frequency. The additional degree of freedom for routing can help to keep the connections short and thus allowing a higher clock speed. A possible application would be to keep normal processing and error handling in different configurations. Since routing can be in particular a bottleneck if multiple devices need to be deployed, dividing an application temporally into multiple configurations can yield in a performance gain compared to a spatial mapping onto multiple devices.

4 Mapping applications onto DRHW

A dynamically reconfigurable architecture alone does not provide the merits described in the previous section. Proper tools are crucial for the successful deployment of DRHW. Today, most hardware designs are implemented on the register-transfer level. At this abstraction level designers are not able to keep pace with the rapidly increasing number of gates provided by modern process technologies. Reconfiguration yields a further degree of freedom that increases design complexity.

4.1 High Level Synthesis

One technique to overcome this productivity gap is to raise the abstraction level to the algorithmic level. High level synthesis divides the sequential control flow of an algorithmic description into several control steps. These control steps are executed sequentially. Within each control step a set of operations of the algorithmic description is executed in parallel. This temporal and spatial mapping makes high level synthesis a good starting point for mapping algorithmic descriptions onto dynamically reconfigurable hardware.

As we have pointed out in the previous section reconfiguration is especially helpful if not all parts of a circuit need to be available at a given instant of time. In order to achieve a high utilization of the silicon area high level synthesis tries to share operators between different control steps. But depending on the application operator sharing is only possible to some degree. Therefore mapping control steps onto different configurations is a good approach to exploit the benefits provided by dynamically reconfigurable architectures.

In [5] NEC has mapped a DES encryption core on their DRLE manually. Although the kernel loop of the encryption reconfigures the DRLE at every clock cycle both performance and energy consumption are improved by more than an order of magnitude compared to a low power embedded microprocessor. This confirms that mapping control steps onto different configurations is a viable approach.

Since commercial tools usually prohibit access to their internal data structures and algorithms we are using our academic high level system CADDY [2]. This enables us to adapt the synthesis algorithms for different dynamically reconfigurable architectures.

4.2 Object Oriented Synthesis

The complexity of today's systems require design on the system level. The object oriented design paradigm enables designers to cope with this complexity. While the software community has been using object oriented techniques successfully for many years object oriented hardware synthesis is still in the domain of academic research.

We have developed an Object Oriented Analysis System (OOAS) that transforms an object oriented description into an equivalent procedural description that is freed from all object oriented constructs [7]. This procedural description can be further processed by a high level synthesis system. The combination of our OOAS and CADDY is therefore a powerful tool set that increases productivity of designers enormously by providing a fully automated design flow from a system level description to a spatially and temporally partitioned description on the register transfer level. This cycle accurate description can be mapped easily onto a dynamically reconfigurable architecture by the place and route tools that need to be provided by the manufacturer of the device.

4.3 Threads

On the system level designers need the ability to describe concurrent behavior by using multiple threads of control. Each thread can be in one of the states *ready*, *running*, or *blocked*. Threads can be blocked because they are waiting for events. Such events can be triggered by an external I/O operation or by another thread. If there is more than one thread ready to run but there are not enough resources to have all of them running in parallel the threads need to be scheduled. I. e., the runnable threads are changing from state ready to running and vice versa.

Those properties of threads are a very good match for reconfigurable architectures: if there is only one thread ready to run this thread can use the entire reconfigurable fabric for itself. If there are multiple threads ready to run the reconfigurable fabric can be shared by swapping configurations in and out (figure 3). If the penalty for reconfiguration is low in terms of latency and power consumption this can be in particular beneficial for accessing relatively slow external memory.

As in the case of workstations where only a limited number of processes or threads can be held in the main memory a dynamically reconfigurable architecture can hold only a limited number of threads. While the workstation would use a

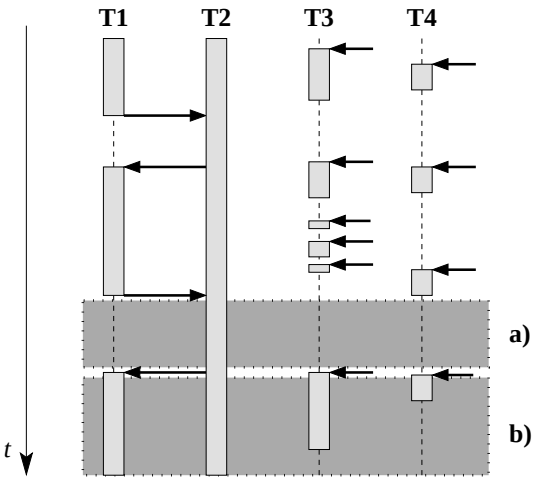

Figure 3. a) T2 is the only thread that is ready to run: it can use the reconfigurable fabric exclusively b) T1-T4 are ready to run: they have to share the reconfigurable fabric

hard drive as an extension of the main memory the next level in the memory hierarchy of a reconfigurable architecture is external memory.

Our OOAS provides the information needed to schedule more threads on the reconfigurable fabric than the device can hold in its internal configuration memory. The main problem is that the internal state of objects must not get lost if a thread is replaced by another thread. The internal state of an object is stored in its attributes. If the attributes are stored in external memory rather than being distributed along the data path there is no need to explicitly save the state of the objects before they are replaced. However, this requires that the execution of all methods is finished at the moment of replacement. This is usually not the case for threads. In order to make methods interruptible the local variables must be saved in addition to the attributes. Storing all variables in external memory is of course not desirable since this would have a strong impact on the performance. It is therefore necessary to find scheduling points where the number of variables that need to be saved is low. These scheduling points must guarantee that the response times of all threads are not violated and that all transactions like accessing external memory are completed.

5 Conclusions

We have defined the term *dynamically reconfigurable hardware* to distinguish it from FPGA-like reconfigurable hardware, and the term *dynamically reconfigurable SoC*. Then we pointed out how the concept of fast reconfiguration can be deployed efficiently in a system level design methodology. Just as the dynamically reconfigurable hardware combines the merits of both the hardware and the software concept we apply proven software techniques to the hardware design flow in order to increase the productivity of IP designers.

6 Acknowledgments

This work is partially supported by DFG project No. RO 1030/8-2 and Robert Bosch GmbH, Germany.

References

[1] Adelante Technologies. http://www.adelantetechnologies.com.

[2] O. Bringmann, W. Rosenstiel, and D. Reichardt. Synchronization detection for multi-process hierarchical synthesis. In *Proceedings of ISSS*, 1998.

[3] Chameleon Systems, Inc. http://www.chameleonsystems.com.

[4] André DeHon and John Wawrzynek. Reconfigurable computing: What, why, and implications for design automation. In *Proceedings of the 36th Design Automation Conference (DAC' 99)*, pages 610–615, New York, June 1999. Association for Computing Machinery.

[5] Koichiro Furuta, Taro Fujii, Masato Motomura, Kazutoshi Wakabayashi, and Msakazu Yamashina. Spatial-temporal mapping of real applications on a dynamically reconfigurable logic engine (DRLE) LSI. In *Proceedings of the 2000 IEEE Custom Integrated Circuits Conference (CICC)*, May 2000.

[6] Reiner Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Proceedings of DATE*, 2001.

[7] Tommy Kuhn, Tobias Oppold, Carsten Schulz-Key, Markus Winterholer, and Wolfgang Rosenstiel. Object oriented hardware synthesis and verification. In *Proceedings of ISSS*, 2001.

[8] PACT XPP Technologies, Inc. http://www.pactcorp.com.

[9] Patrick Schaumont, Ingrid Verbauwhede, Kurt Keutzer, and Majid Sarrafzadeh. A quick safari through the reconfiguration jungle. In *Proceedings of the 2001 Design Automation Conference (DAC-01)*, pages 172–177, New York, June 18–22 2001. ACM Press.

[10] Tensilica. http://www.tensilica.com.

[11] Wind River Systems, Inc. http://www.windriver.com.