Int J Parallel Prog
DOI 10.1007/s10766-009-0119-4

A Self Distributing Virtual Machine for Adaptive Multicore Environments

Jan Haase · Andreas Hofmann · Klaus Waldschmidt

Received: 25 February 2008 / Accepted: 23 September 2009

Abstract The use of parallel systems is no longer limited to dedicated clusters as multicore chips are more and more appearing in embedded applications. To meet power, performance and cost targets these systems need to be adaptive. The reconfiguration features of recent FPGAs make new approaches for this type of parallel computing possible: Dynamic reconfiguration at runtime offers an important step to adaptive behavior of systems-on-chip (SoCs). This article analyzes the challenges of such an adaptive SoC. It is shown that many of the requirements for an adaptive FPGA-realization are met by the SDVM, the scalable dataflow-driven virtual machine which has been successfully implemented and tested on a cluster of workstations. The SDVM has evolved to a virtualization layer for multicore-FPGAs, now called SDVM^R. This virtualization layer allows a transparent runtime-reconfiguration of the underlying hardware to adapt to the changing system environment. Results for a basic application for both systems are presented.

Keywords Adaptive systems · Multicore systems · FPGA · Virtualization

J. Haase (✉)
Institute of Computer Technology, Technical University Vienna, Gußhausstr. 27-29/384,
1040 Vienna, Austria
e-mail: haase@ict.tuwien.ac.at

A. Hofmann · K. Waldschmidt
Technical Computer Sc. Dep., J.W. Goethe-University, Box 11 19 32, 60054 Frankfurt, Germany
e-mail: ahofmann@ti.cs.uni-frankfurt.de

K. Waldschmidt
e-mail: waldsch@ti.cs.uni-frankfurt.de

Published online: 21 November 2009

1 Introduction

In the beginning parallel systems were implemented as dedicated clusters. The rapidly increasing number of transistors per chip enables the integration of multiple cores which promises a significant speed-up if the parallelism of the application can be fully exploited. So, these days parallel systems more and more consist of multicore processors, multicore embedded systems, or even multicore FPGA-based devices.

As environmental parameters change frequently and sometimes fast, especially for embedded systems, a static configuration is disadvantageous. Multicore systems offer new degrees of freedom to adapt to the changing environment at runtime if each core—whether it is a general-purpose or an application specific one—can be configured individually and tasks can be shifted between the cores. Configuration by hand of such a dynamically changing system is hard or even impossible. Thus the adaptivity should be managed by the system itself autonomously. Techniques to implement the adaptivity like self-diagnosis, self-configuration, and self-optimization are currently under development and known from the subject of biologically inspired or organic computing [1].

Modern platform FPGAs featuring multiple processor cores and the ability to reconfigure themselves at runtime provide a good basis to develop such adaptive systems. However, the long familiar problems of performance, reliability, flexibility, and power management still exist in FPGAs. To efficiently use multicore FPGAs a parallel system must be created which includes every core. To optimize the power management the number of active, or even configured, cores must be adapted dynamically to the current workload. Furthermore, the configurable logic has to be used to implement application specific function units to accelerate performance.

To make these features of an FPGA manageable a software model is needed, which hides the—due to runtime reconfiguration—changing hardware system from the application software. The scalable dataflow-driven virtual machine (SDVM) is such a virtualization of a parallel, adaptive and heterogeneous cluster of processing elements [2,3]. Thus, it is well suited to serve as a virtualization layer for multicore FPGAs. The FPGA virtualization layer, called SDVM^R, is currently under development.

This article shows the evolution of the virtualization layer for a cluster environment, the SDVM, to the software model for dynamic reconfigurable Multicore FPGAs, the SDVM^R. Section 2 describes the concept and realization of the cluster virtualization layer. Section 3 analyzes the basic requirements of multicore systems in general and dynamic reconfigurable FPGAs in particular that shape the development of the SDVM^R. Section 4 shows some results for a simple application, namely the Romberg numerical integration algorithm, both for the cluster and the FPGA-version of the SDVM. The article ends with a conclusion in Sect. 5.

1.1 Related Work

The usage of adaptive features to tackle the complexity of modern systems-on-chip incorporating multiple cores is extensively covered by Lipsa et al. [4]. Their paper proposes a concept that applies autonomic or organic computing principles to hardware

designs. The paper does not present any kind of implementation, neither as software nor as hardware. The SDVM as a virtual machine for FPGAs is a software-realization of these autonomous principles based on today's FPGA technology.

Lysecky et al. [5] developed techniques for dynamic hardware/software partitioning based on on-line profiling of software loops and just-in-time synthesis of hardware components called WARP. They also present a dynamic FPGA routing approach which can be used to solve the routing and placement problem of reconfigurable components at runtime [6]. However, their approach relies on a special, to our knowledge not yet implemented, FPGA architecture called configurable logic architecture [7]. In contrast, the virtualization layer implementation presented in our article targets existing FPGA hardware namely the Xilinx Virtex-4 families.

2 The Scalable Dataflow-Driven Virtual Machine (SDVM)

The scalable dataflow driven virtual machine (SDVM) [2,3] is a dataflow driven parallel computing middleware (see Fig. 2). It was designed to feature undisturbed parallel computation flow while adding and removing processing units from computing clusters (i.e. a gang of interconnected computing devices). Applications for the SDVM must be cut to convenient code fragments (of any size). The code fragments ("microthreads") and the microframes (a data container for parameters needed to execute them, see Fig. 1) will be spread automatically throughout the cluster depending on the data distribution [3].

Each processing unit which is encapsulated by the SDVM virtualization layer and thus acts as an autonomous member of the cluster is called a *site*. The sites consist of a number of modules (managers) with distinct tasks (see Fig. 3) and communicate by message passing.

Currently the SDVM is implemented as a prototypical UNIX-daemon to be run on each participating machine or processor, creating a site each [8].

2.1 The SDVM Daemon

The SDVM daemon consists of several managers with different fields of responsibility. Some deal with the execution of code fragments, some attend to communications with other sites, some are concerned with the actual decision-making (see Fig. 3).

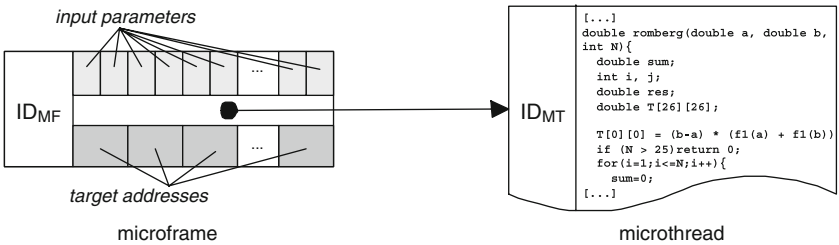

Fig. 1 Microframes contain a pointer to a microthread and space for all parameters needed for its execution

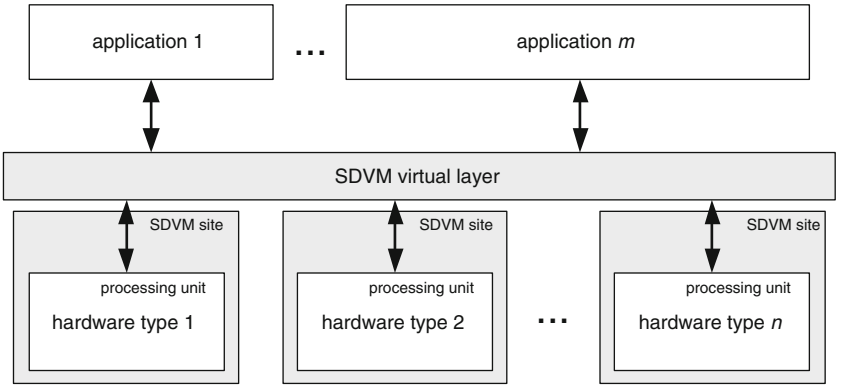

Fig. 2 The processing units are encapsulated by an SDVM site each. The sites form the SDVM virtualization layer. The applications do not see the underlying (possibly heterogeneous) hardware

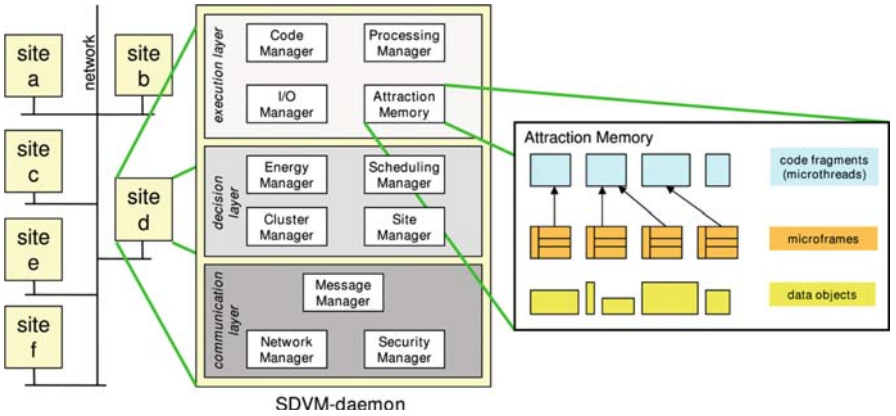

Fig. 3 The cluster consists of a number of sites connected over any kind of network. Each site in turn is composed of a number of modules (called managers) with distinct tasks. The attraction memory, for example, manages the microframes which hold the parameters to execute the corresponding code fragment

2.1.1 The Execution Layer

The execution layer is responsible for the handling and execution of the code and data. Furthermore, it provides I/O virtualization.

Microframes waiting for more parameters as well as global memory objects are kept in a so-called attraction memory. If a data object is requested, it is first sought locally. In case of a miss the site it actually resides on is determined and then the data object is moved or copied to the local site.

Microthreads are only requested when they are to be executed locally. The local caching of microthreads and the compilation of microthreads, if needed, is done by the code manager.

The processing manager executes the microthread/microframe pair. To accomplish this, it provides an interface for the microthread to read the parameters of its microframe. When the execution has finished the processing manager deletes the no longer needed microframe. To hide network latencies when e.g. an access to a remote part of the global memory is needed, the processing manager may execute several microthread/microframe pairs concurrently.

The input/output manager manages user interaction and accesses local resources like hard disks or printers.

2.1.2 The Communication Layer

The communication layer manages sending and receiving of messages between sites. The message manager is the central communication hub for all other managers. It generates serialized data packets to be sent to other sites, adds information about the local site and determines its address before optionally passing them to the security manager. This manager may then encrypt and sign the data packets to avoid e.g. eavesdropping and spoofing. On the receiving site it will validate the signature and decrypt the message, if necessary, before passing it to the message manager.

The network manager is the part of the SDVM which is responsible for the actual transportation of the data packets. For the currently existing cluster realization it uses TCP/IP to send data to other sites. For an implementation of the SDVM on multiprocessor chips it would have to use the on-chip network to pass data to the receiving site.

2.1.3 The Decision Layer

While the responsibilities of the managers in the execution and communication layers are more or less usual in computer systems, the decision layer implements the more sophisticated parts and potential self-x-properties of the SDVM.

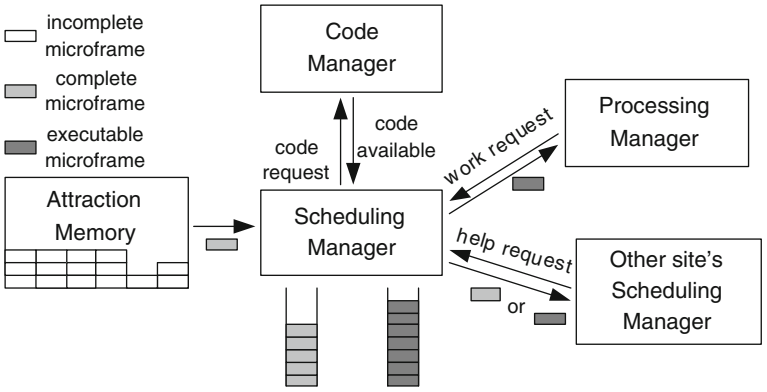

Fig. 4 The SDVM's scheduling concept. Incomplete microthreads still wait for certain parameters. Complete microframes contain all needed parameters but the corresponding microthread is still missing. When the microthread is received, the microframe becomes executable

The SDVM features distributed scheduling which is done by the scheduling manager (see Fig. 4): Incomplete microframes are stored in the Attraction memory, until all needed parameters were received and filled in. The completed microframes (having all needed parameters but the corresponding microthread is still missing) are given to the Scheduling Manager which triggers the Code Manager to fetch the missing microthread. When the microthread is received, the microframe becomes executable. The site's own Processing Manager then requests work and receives an executable microframe. If the Scheduling Manager can not provide work, it requests microframes from other sites (help request).

Most scheduling methods assume a central calculation of the execution order, combined with a centrally managed load balancing. They take advantage of the accord that all information is collected on one site and thus good scheduling decisions can be made. However, in big clusters this central machine may become a bottleneck or even a single point of failure.

The SDVM works without client-server concepts as far as possible. Therefore, the scheduling is done autonomously by each site. The sites do not have knowledge about the current global execution status of the application, but only about the locally available executable microframes. Some information can be extracted in advance, though: The dataflow graph of the application contains all microthreads and therefore the critical path of an application and regions of high data dependencies can be detected. These parts will then be executed with higher priority resp. executed preferably on the same site.

The site manager and the cluster manager collect and distribute data about the local site and the whole cluster, respectively. The collected data provides the basis for the algorithms that implement the self-x properties of the SDVM. One example is the energy manager which controls the energy state of the local site. The energy manager sets the local energy state according to a given energy management policy depending on factors like current and past resource utilization, and local temperature. This can be used to influence the reliability of the system [9].

2.2 Adaptation of the SDVM to Platform FPGA

The SDVM is a convenient approach as a middleware (virtualization layer) for FPGAs due to several distinguishing features. These features include:

- undisturbed parallel computation while resizing the system
- dynamic scheduling and thereby automatic load balancing
- distributed scheduling: no specific site has to decide a global scheduling and therefore any site can be shut down at any time
- participating computing resources may have different processing speeds
- participating computing resources may have different instruction sets (e.g. different ISAs for hardcores and softcores), as matching precompiled code fragments are used automatically
- applications may be run on any SDVM-driven system, as the number and types of the processing units do not matter

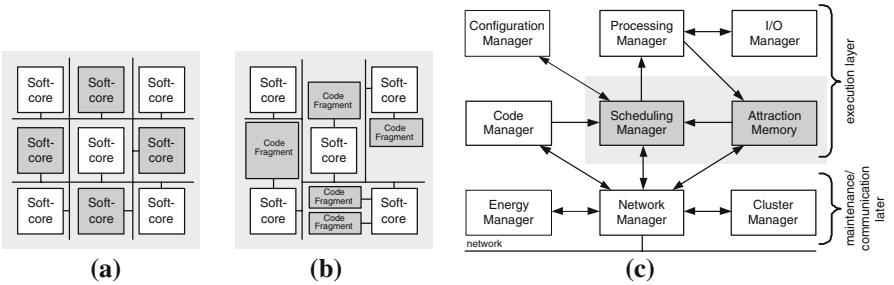

Fig. 5 Different possibilities to use the FPGA's resources. **a** Create more softcores; **b** Realize some code fragments in hardware; **c** Realize some managers in hardware (depicted in *grey*)

- support for any connection network topology
- no common clock needed: the clock is locally synchronous but globally asynchronous.

The SDVM is currently being ported to run on an FPGA. There are three possibilities currently investigated to use the reconfigurable area provided by the FPGA.

(1) The available resources on the FPGA are used up by configuring additional processing units (softcores). Thus the SDVM cluster consists of more sites and a higher parallelism can be achieved. (See Fig. 5a)
(2) The FPGA fabric is used to implement custom function units, each attached to and therefore controlled by one of the cores. The function units conform to specific code fragments which are to be executed often. The execution is then done in hardware and therefore much faster. The supported functions of the custom function units can be changed at runtime by reconfiguration. In this case the execution of the applications is accelerated by hardware implementation. (See Fig. 5b)
(3) Each SDVM site consists of several modules which deal with e.g. communication, scheduling, or cluster management. Some of the calculation intensive modules are realized in hardware by configuration. (Some of) the sites forming the SDVM cluster will therefore run faster. In this case the SDVM-layer is accelerated by hardware implementation. (See Fig. 5c)

These different approaches can be combined for optimization.

3 Adaptive Multi-Core Environments

The concept of the SDVM has been successfully implemented and tested on a cluster of workstations. Based on the results gathered in this environment, the SDVM has been proposed as a promising concept to be used for dynamically reconfigurable systems [10]. To back up this claim this Section explores the requirements of such systems and shows how the features of the SDVM match. Based on these requirements the SDVM has evolved to a virtualization layer for Multicore-FPGAs, now called SDVM^R [11].

3.1 Fundamental Design Challenges

Multicore systems can be classified in three major categories. The first and most traditional category covers multicore systems which are built out of multiple instances of a certain core architecture. Typical examples are multicore processors readily available from Intel or AMD. These multicores feature limited adaptivity usually restricted to voltage and frequency scaling individually for each core.

The next class consists of systems-on-chip (SoCs). The fundamental difference to the aforementioned class is their feature to incorporate heterogeneous cores like generic processors, digital signal processors and special functions units. Adaptivity in these systems not only covers voltage and frequency scaling but can also include task relocation between functional cores of different architecture. One example of such a system is the Texas Instruments OMAP controller family [12] targeted at mobile communication and multimedia appliances. Each member of this family contains a general-purpose microcontroller core accompanied by one or more digital signal processors together with application specific cores.

The third class includes field programmable gate arrays (FPGAs) which can be used to implement SoCs or straight multicore CPUs and further augment these systems with reconfigurability that provides additional flexibility to the system.

Besides the primary functions that a system-on-chip (SoC) should accomplish, e.g. speech encoding in a cell phone, their design has to address a multitude of secondary requirements. These requirements are important for most systems, merely the weighting differs. The introduction of FPGAs as a target platform for SoCs adds an other important requirement: The runtime reconfiguration ability. To make optimal use of these reconfigurable systems an efficient management of the reconfiguration process is necessary.

Thus, the design of multicore systems and their software must address the following challenges:

- performance
- scalability
- incorporation of heterogeneous components
- adaptivity
- support for parallelism
- robustness and reliability
- energy efficiency
- reconfigurability

As these requirements and therefore the techniques to achieve them are common to all classes of multicore systems it is beneficial to supply a generic module which manages these supporting features. This lightens the burden of the designer who can concentrate on the primary functions of the SoC. Our discussion in this article focuses on dynamically reconfigurable FPGAs as an implementation architecture for multicore systems.

The generic module should be implemented as a functional layer between the system hardware and the application software thus acting as a middleware. To avoid an

increase in complexity, provide flexibility, and improve portability and code reusability through different hardware types the division into several layers is a possible solution.

The middleware should provide a complete virtualization of the underlying hardware. The application has no longer to be tailored to the hardware, instead it is sufficient to tailor it to the virtualization layer. This virtualization layer not only provides hardware independence, it can also hide changes in the underlying hardware due to reconfiguration. Thus such a middleware is specifically well suited to be used as a virtualization layer for FPGAs or adaptive multicore chips.

3.2 Dynamically Reconfigurable Platform FPGAs

Modern platform FPGAs augment the logic fabric, which consists mainly of configurable logic blocks (CLBs) and a routing network, with a number of function blocks. The members of a certain FPGA family differ in the size of the CLB array and the number and types of special function blocks. These function blocks include:

- processor cores e.g. PowerPC
- embedded memory blocks
- multiplier and basic DSP (digital signal processing) blocks
- communication interfaces adhering to various standards like ethernet or PCIe (peripheral component interconnect (PCI) express)

Any function block which is implemented on an FPGA using dedicated silicon area and therefore using no CLBs is called a *hard macro*. If the function block is a processor core it is more accurately called a *hardcore*. In contrast, processor cores which are implemented solely using the CLBs are called *softcores*. However, hardcores may require additional support logic that has to be implemented in CLBs to be fully usable.

The vast amount of configurable logic blocks enables the designer to add several softcores. As seen in Table 1 even the second smallest device of the Virtex-4 FX family can host four MicroBlaze softcores [13] and still has more than 60% free logic resources that can be used to implement application specific functions.

3.3 Dynamic Partial Reconfiguration

Reconfiguring FPGAs at runtime offers a number of valuable benefits which facilitates new areas of application. The main benefits include:

- The resource utilization can be increased if the FPGA area dedicated to some currently unused module is reassigned. So, ideally the FPGA's size does not have to

Table 1 Resource requirements of MicroBlaze based multi-processor systems implemented on a Xilinx Virtex-4 FX20

System	4-Input LUTs	DSP48 blocks
1 MicroBlaze	1,275 (14%)	3 (9%)
2 MicroBlaze	2,889 (16%)	6 (18%)
4 MicroBlaze	5,487 (32%)	12 (37%)

be as large as to host all modules which are used eventually; the size can be limited to the largest working set required.
- Hardware can be shared between various applications. Each application is supported by specialized hardware modules which are reconfigured when needed.
- Hardware can be updated remotely without the need to shutdown the system. Furthermore, the number of external components can be greatly reduced as there is no need for external logic to do the configuration update.
- Algorithms like pattern matching can map their patterns onto the logic fabric while maintaining the ability to update the patterns at any time. Mapping such short-time constant data onto the logic fabric can improve performance as the number of memory accesses is reduced.
- Soft-errors in the FPGA configuration can be corrected during runtime. The device configuration can be continuously read back and checked via embedded ECC bits. In case of a bit flip the corrected configuration can be restored by reconfiguration of the affected part of the FPGA.

Reconfiguration can be done in two ways: The whole logic fabric or only some part of it is reconfigured. The former can only be done by some external hardware whereas the latter enables the FPGA to reconfigure itself if some of the logic fabric can continue to run during reconfiguration.

The benefits of reconfiguration can be efficiently exploited if the FPGA:

- can be reconfigured an unlimited number of times
- has a short configuration time in the order of magnitude of a memory access
- is able to reconfigure itself
- can continue to operate the uninvolved part of the logic fabric while reconfiguring another part
- offers fine grained configuration of its logic fabric
- offers glitch free transition between the old and the new configuration

FPGAs based on SRAM cells to store the current configuration can be reconfigured an unlimited number of times. Furthermore, in theory configuration delays can be quite low because at heart a configuration change is just a write access to a number of SRAM cells. So, each SRAM based FPGA—by far the most widespread type—features the basic attributes to support runtime reconfiguration.

However, most FPGA architectures, even the SRAM based ones, are not designed with a major focus on runtime partial reconfiguration. To improve logic density and thus increase performance and reduce cost, the SRAM cells of an FPGA are chained together to form a shift-register with up to 54,000,000 bits [14]. This is beneficial for initial configuration as the FPGA can be fed with a serial bitstream but it poses a twofold difficulty as reconfiguration can be neither fast nor fine grained. Furthermore, the SRAM cells which hold the FPGA configuration are optimized for low leakage instead of access speed.

To support partial reconfiguration in spite of the aforementioned properties the actual implementation of the configuration memory is slightly different. The configuration memory of the Xilinx Virtex-4 family is arranged in frames with a size of 1,312 bits each. Every frame corresponds to a small tile of the FPGA consisting of 16 adjacent CLBs, a couple of IOBs, or some block RAM. Each frame can be

independently read and written, thus moderate runtime reconfiguration granularity is possible.

The internal configuration interface of the Virtex-4 FPGAs has a maximum clock frequency rating of 100 MHz. Thus, the readback of one frame requires about 1.6 μs. A read-modify-write operation for one frame takes about 30 μs. All in all changing the reconfiguration memory needs about one to two magnitudes more time than accessing FPGA block RAM. Although, it still seems fast enough to be useful for thread-based runtime reconfiguration as typical operating systems switch threads every couple of milliseconds.

The new members of the Xilinx Virtex family, namely the Virtex-5 and Virtex-6, retain the general architecture regarding dynamic reconfiguration.

3.4 The Virtualization Layer Concept

Today, even small FPGAs can host multiple processing elements (See Table 1). A good choice for the architecture of these processing elements (PE) would be the MicroBlaze softcore for implementation in the CLBs or the PowerPC hardcore which is present on some FPGA families. However, even special purpose function blocks like digital signal processors (DSP) or application specific blocks can be used.

Therefore, one of the fundamental decisions in the design process of the virtual machine is whether each PE is to form an independent building block of the parallel cluster or multiple PEs are merged in a higher-order cluster element. The latter may impose less overhead but the former eases the implementation of adaptive features; in detail:

- coping with errors in the fabric
- avoiding bottlenecks
- reducing hot spots.

If each processing element is augmented with a complete set of the virtualization functions and therefore no PE is the sole provider of any function, the system is much more flexible. If an error is detected in some part of the FPGA the affected PE can be disabled or reconfigured to avoid the erroneous location without hampering the functionality of the cluster. Furthermore, as each augmented PE provides its share of the cluster management functionality the number of bottlenecks is reduced. The distribution of functionality can lead to a better distribution of workload thus reducing the number of hot spots on the FPGA.

The logic resources and therefore the computing power of the FPGA and the internal memory blocks can be distributed evenly among all processing elements, but there are resources which cannot be efficiently split. The most important one being the external memory. As FPGAs typically contain only up to some hundred kilobytes of internal memory—the smaller ones actually provide less than one hundred kilobytes—a lot of applications require external memory. Therefore, the middleware should support a multi-level memory architecture that is transparent to the application software.

Besides external memory every interface of the FPGA system to the outside world like ethernet or PCIe cannot be allocated to every PE concurrently. The middleware must manage these resources on the cluster level.

The middleware should provide a complete virtualization of dynamically reconfigurable platform FPGAs, so that the user application does not need to care about the underlying hardware. Therefore, it has to support the following primary features:

- Combine all processing elements (PE) on the FPGA to create a parallel system
- Provide task mobility between all processing elements even if they are heterogeneous. It should be possible to execute a task on general purpose processors of different architectures and on custom function units if applicable.
- Virtualize the I/O-system to enable the execution of a task on an arbitrary processing element
- Combine the distributed memory of each PE to form a virtually shared memory. To avoid bottlenecks each PE should have its own memory both for program and data but, as applications for shared memory are much easier to design than applications for message passing systems, the memory should appear globally shared to the application.
- Manage the reconfiguration of the FPGA, i.e. keep track of the current usage of the FPGA resources and available alternative partial configurations. Furthermore, an adequate replacement policy has to be defined.
- Monitor a number of system parameters to gather information the configuration replacement policy depends on
- Adjust the number of active processing elements at runtime. For example, this can be used to meet power dissipation or reliability targets.
- The previous feature requires the middleware to hide the actual number of processing elements from the application to ease programming.
- As the user software does not know the number and architecture of the processing elements the middleware has to provide dynamic scheduling as well as automatic code and data distribution.
- The execution of the workload should not be disturbed by any reconfiguration activities. The execution performance should scale with the number of active components.

The implementation of these primary features is greatly helped by the microframe/microthread concept of the SDVM (see Fig. 1). As the microthreads are ready for execution when the associated microframe got all its data the application programmer does not need to know the number of active processing elements. Accessing microframes whether they are local or stored on a distant site is transparently managed by the attraction memory thereby providing a virtual shared memory. As the SDVM distributes microframes and microthreads automatically throughout the system and acts as a hardware abstraction layer it facilitates task mobility.

The current state of the application is stored in the set of currently allocated microframes. As long as this set is preserved and running microthreads are not interrupted the number of processing elements can be changed at any time.

In the beginning these features are provided in software by the SDVM^R. The development stage of the SDVM^R—whose performance results are presented in

Sect. 4—supports the inclusion of processing elements at runtime, a virtual shared memory, dynamic scheduling and a virtualized I/O system based on memory-mapped I/O. Task mobility is limited to the case that microframes could be transferred to any processing element. Microthreads are expected to be initially present at any processing element by statically linking them with the SDVM^R code.

In the future, it is planned to investigate the feasibility of a hardware implementation.

4 Results

Both the SDVM and the SDVM^R need a lot of calculations and communication to orchestrate the distributed execution. Therefore, a question is whether the additional overhead is small enough to maintain the concept.

For demonstration and evaluation, the Romberg numerical integration algorithm [15] was implemented on both systems.

This algorithm partitions the area to be calculated into several portions of constant width (Frame 1). Those can be calculated independently and the results added eventually. The first microthread will generate a target microframe (Frame 3) where the results are finally added and then, in our example, 100 or 150 other microframes (Frame 2) are generated, which can be run in parallel. Fig. 6 shows this behavior in form of a control-dataflow graph (CDAG [16]).

4.1 SDVM—UNIX-Based Cluster Implementation

A test bench was implemented by using a cluster consisting of four similar Intel PCs. Each site in the test bench can simulate one core of a multicore FPGA or multi-processor system (see Fig. 7).

First, it shall be demonstrated how much overhead is generated by using the SDVM. To show this, run times on a stand-alone SDVM site are compared with the run times of a corresponding sequential program (see Fig. 8). This overhead appears to be about 2%, even if the microthreads have to be compiled before execution.

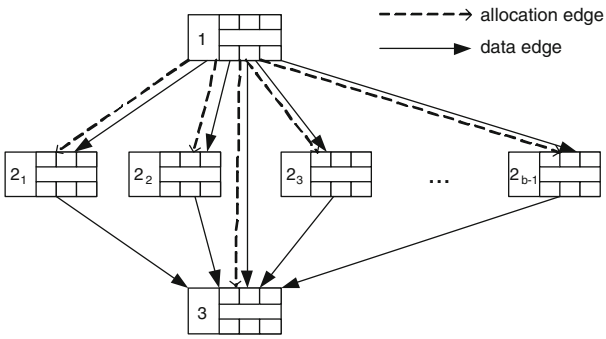

Fig. 6 The CDAG of the Romberg example application

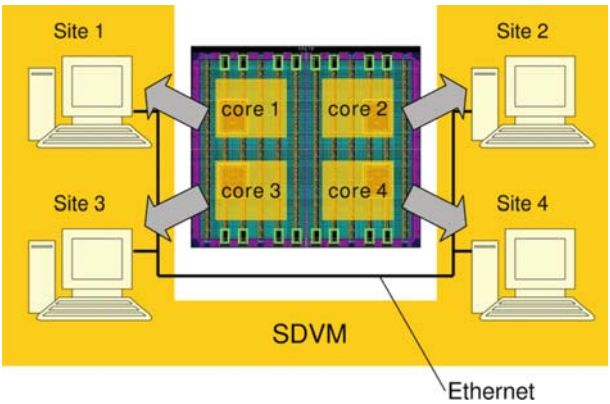

Fig. 7 Four similar Intel PCs are used to simulate a quad-core system

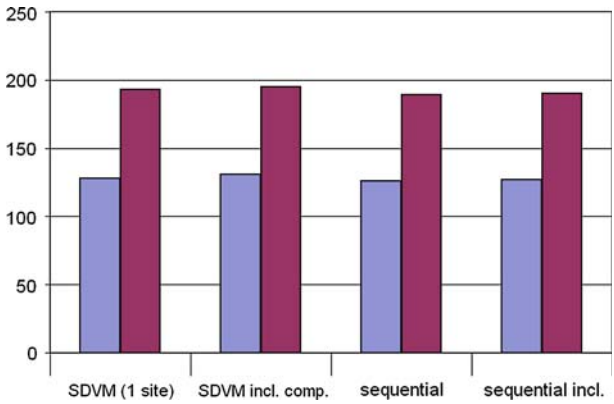

Fig. 8 Romberg algorithm: comparison of the run times (in seconds) of a sequential program and the SDVM with one site. Values are given with and without compilation time, respectively, for width 100 and 150

In the next step, it has to be shown that the speedup is in expected regions. On a cluster of identical machines (Pentium 4, 1.7 GHz), a value for the speedup is shown in Fig. 9. It reaches roughly the number of participating sites, which is a good result (see Table 2).

4.2 SDVM^R—FPGA-based Multicore Implementation

To evaluate the virtualization layer concept for Multicore-FPGAs a scalable system has been created using the Xilinx EDK 10.1 software (See Fig. 10). As the hardware basis a Virtex-4 FX20 populated evaluation board [17] was chosen due to its fine-grained reconfiguration features and embedded PowerPC core.

The system is based on IP blocks supplied by Xilinx. Besides the PowerPC405 embedded in the Xilinx Virtex-4 MicroBlaze softcores are used as processing

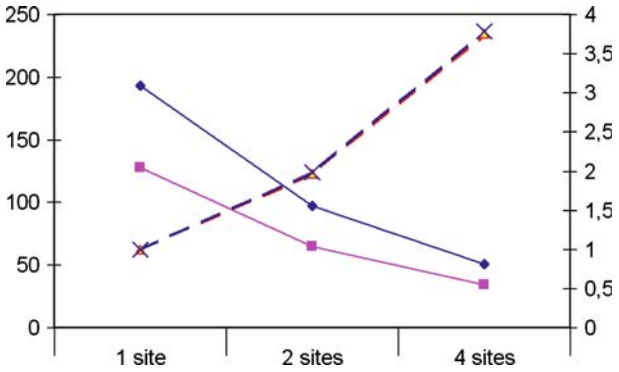

Fig. 9 Romberg algorithm: run times (in seconds) and speedup depending on the number of sites

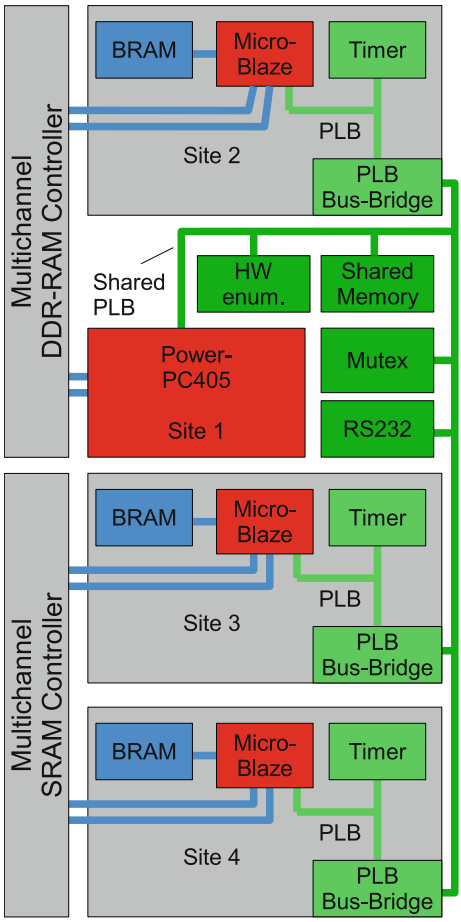

Fig. 10 The system implemented on a Xilinx Virtex-4 FX20. The debug module connected to each MicroBlaze and the PowerPC core is not shown for simplicity

Table 2 Romberg algorithm: run times (in seconds) and speedup depending on the number of sites

	1 Site	2 Sites	4 Sites
Width 100	128	65	34
Width 150	193	97	51
Speedup width 100	1	1.97	3.76
Speedup width 150	1	1.99	3.78

Table 3 Size of ELF file of the stand-alone Romberg and SDVM^R version

CPU	Stand-alone (KiB)	SDVM^R (KiB)	SDVM^R overhead (KiB)
PPC	109	191	82
MBlaze	99	192	93

The file includes all code and data segments including 48 KiByte thread stack space. About 1 KiB of dynamically allocated memory is required for the Romberg algorithm which is excluded here

elements. The MicroBlaze is supported by a timer connected to a local Processor Local Bus (PLB) to allow for the execution of the Xilinx XMK operating system. Each MicroBlaze core has 16 KiB of embedded memory blocks connected to its Local Memory Bus (LMB).

As can bee seen in Table 3 even the simple Romberg application requires about 100 KiByte memory. A significant part of it is claimed by the Xilinx microkernel configured with support for 5 user threads. Thus more memory than what is available as block RAM on the FPGA is required. Therefore, external 128 MiB DDR-RAM and 1 MiB SRAM is connected to the MicroBlaze and PowerPC cores using multichannel memory controllers. Each MicroBlaze has 8 KiB data and 8 KiB code cache to speed up external memory access. The MicroBlaze is implemented with all features enabled except FPU. Performance of the MicroBlaze is somewhat reduced due to the necessity to use the area optimized version of the version 7.1 core.

The communication between the processing elements is done using a shared memory connected to the system PLB. To allow for mutual exclusion of the access to this memory and to the RS232 interface a hardware mutex module is attached to the system PLB. The multiple cores are attached to the system using PLB-to-PLB bridges.

The busses and MicroBlaze cores run at 100 MHz clock frequency; the PowerPC core is clocked at 300 MHz. The system occupies 94% of the FPGA slices and 97% of on-chip memory blocks.

Table 4 Romberg algorithm: sequential calculation: run times (in seconds) for one Romberg integration of width 150 on an FPGA with one PowerPC (site 1) and three MicroBlaze cores (sites 2, 3, and 4)

	Stand-alone (s)	Relative to PPC	SDVM^R (s)	Relative performance (%)
Site 1	41.43	1	42.67	97.04
Site 2	251.27	0.16488	251.65	99.85
Site 3	241.07	0.17186	239.64	100.5
Site 4	246.84	0.16784	244.53	100.9

Four instances of the application were run in parallel, one on each core

Table 5 Romberg algorithm: parallel calculation: run times (in seconds) for one Romberg integration of width 150 on an FPGA with one PowerPC and three MicroBlaze cores

	Stand-alone (theoretical) (s)	Relative to PPC	SDVM^R (s)	Relative performance (%)
Average	27.54	1.50458	30.16	91.3
Peak	27.54	1.50458	29.38	93.7

The SDVM^R was run as a cluster using all four cores. The stand-alone value is the theoretical peak performance based on per-core run times (See Table 4)

To demonstrate the overhead run times of the stand-alone and the SDVM^R version of the Romberg integration are compared (See Table 4). These values are not comparable to the ones measured for the SDVM cluster implementation as the iteration depth of the Romberg algorithm is different to achieve reasonable run times. Without an FPU to support the double precision Romberg integration the cores of the FPGA-based system are about 200 times slower than the Intel Pentium-4 CPUs.

The applications were run independently on each core but at the same time. Thus all shared resources like busses and memory controllers were roughly utilized like in a real parallel application running distributed on all four cores. On the PowerPC core the SDVM^R overhead is about 3%. On the MicroBlaze cores the run times show no overhead. The SDVM^R is even slightly faster than the stand-alone Romberg application. This is not to be expected as the PowerPC run times show that there is clearly some overhead involved. The slightly better performance on the MicroBlaze cores may be attributed to different memory access patterns of the SDVM^R version that suit the memory and cache controllers better. The run times of all sequential calculations were consistent over all iterations.

The theoretical peak performance for the system is calculated based on the run times measured for the stand-alone sequential Romberg integration. The measured values for each core have been normalized by computing the relative performance of each and relating them to the performance of the fastest core, the PowerPC. The theoretical peak performance is the sum of these normalized values. Thus a perfectly parallel and distributed Romberg integration would run for 27.54 s on this system.

The SDVM^R running the Romberg integration distributed on all four cores yielded an average run time of 30.16 s. Run times differed for each run due to the dynamic scheduling of the SDVM^R. A peak performance of 29.38 s has been recorded. Therefore, the current implementation of the SDVM^R achieves 91.3–93.7% of the theoretical peak performance. This is a good result given that the performance of the cores differs greatly without the scheduling algorithm explicitly taking this into account (Table 5).

5 Conclusion

In this article a virtualization layer for multicore environments, especially FPGAs, was presented which separates applications to be run from the underlying hardware. It is based on the SDVM, a middleware for computer clusters and multicore chips

which has been successfully implemented and tested on a cluster of workstations. The prototype and its full source code is freely downloadable [8].

The SDVM can behave self-organizing as sites may join or leave at runtime without disturbing the execution of running applications, the cluster may grow or shrink to any convenient size, moreover regardless of the sites' hardware or the network topology between them. The cluster scales automatically.

It is self-optimizing as it automatically distributes data and program code to sites where it is needed, thereby dynamically balancing the workload of the whole system.

Based on the results gathered in this environment, the SDVM has evolved to a virtualization layer for Multicore-FPGAs, now called SDVM^R. Due to the SDVM^R's features, the FPGA may reconfigure itself at runtime to adapt to changing conditions and requirements. The adaptivity of the system is supported by the ability of the SDVM^R to act self-organizing and self-optimizing. Preliminary performance results are given. These results show that the degree of parallelism present in the example application could be exploited by the SDVM^R to a great extent.

In the future, it is planned to investigate the feasibility of a hardware implementation.

References

1. VDE/ITG/GI-Arbeitsgruppe Organic Computing: Organic Computing, Computer- und Systemarchitektur im Jahr 2010. Technical report, VDE/ITG/GI (2003) (in German)
2. Haase, J., Eschmann, F., Waldschmidt, K.: The SDVM—an approach for future adaptive computer clusters. In: 10th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS 05), Denver, Colorado, USA (2005) http://www.ti.cs.uni-frankfurt.de/parallel/papers/DPDNS05.pdf
3. Haase, J., Eschmann, F., Klauer, B., Waldschmidt, K.: The SDVM: a self distributing virtual machine. In: Organic and Pervasive Computing—ARCS 2004: International Conference on Architecture of Computing Systems. Volume 2981 of Lecture Notes in Computer Science, Heidelberg, Springer Verlag (2004) http://www.ti.cs.uni-frankfurt.de/parallel/papers/ARCS04.pdf
4. Lipsa, G., Herkersdorf, A., Rosenstiel, W., Bringmann, O., Stechele, W. : Towards a framework and a design methodology for autonomous soc. In: Brinkschulte, U., Becker, J., Fey, D., Hochberger, C., Martinetz, T., Müller-Schloer, C., Schmeck, H., Ungerer, T., Würtz, R.P. (eds.) ARCS Workshops, pp. 101–108. VDE, Verlag (2005)
5. Lysecky, R., Vahid, F.: A study of the speedups and competitiveness of fpga soft processor cores using dynamic hardware/software partitioning. In: DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe, Washington, DC, USA, IEEE Computer Society, pp. 18–23 (2005)
6. Lysecky, R., Vahid, F., Tan, S.X.D.: Dynamic fpga routing for just-in-time fpga compilation. In: DAC '04: Proceedings of the 41st Annual Conference on Design Automation, New York, NY, USA, ACM Press, pp. 954–959 (2004)
7. Lysecky, R., Vahid, F.: A configurable logic architecture for dynamic hardware/software partitioning. In: DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe, Washington, DC, USA, IEEE Computer Society, p. 10480 (2004)
8. Haase, J.: The Self Distributing Virtual Machine—Homepage (2005) http://sdvm.ti.cs.uni-frankfurt.de.
9. Haase, J., Damm, M., Hauser, D., Waldschmidt, K.: Reliability-aware power management of multi-core processors (2006) DIPES 2006, Braga, Portugal
10. Haase, J., Hofmann, A., Waldschmidt, K.: The self distributing virtual machine (SDVM): making computer clusters adaptive. In: Biologically Inspired Cooperative Computing. Number 216/2006 in IFIP International Federation for Information Processing, Springer Boston, pp. 169–178 (2006)
11. Hofmann, A., Waldschmidt, K.: SDVM-R: a scalable firmware for FPGA-based multi-core systems-on-chip. In: ISVLSI, IEEE Computer Society, pp. 387–392 (2008)

12. Texas Instruments: OMAPV1035 Product Bulletin (2006) http://focus.ti.com/pdfs/wtbu/TI_omapv1035.pdf
13. Xilinx: MicroBlaze Processor Reference Guide (2008) http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf
14. Xilinx: Virtex-4 Configuration Guide (2006) http://www.xilinx.com/bvdocs/userguides/ug071.pdf
15. Dahlquist, G., Bjorck, A.: Numerical Methods. Prentice Hall, Englewood Cliffs (1974)
16. Klauer, B., Eschmann, F., Moore, R., Waldschmidt, K.: The CDAG: a data structure for automatic parallelization for a multithreaded architecture. In: Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP 2002), Canary Islands, Spain, IEEE (2002)
17. Xilinx: ML405 Evaluation Platform: User Guide (2008) http://www.xilinx.com/support/documentation/boards_and_kits/ug210.pdf