R. W. Hartenstein, R. Kress, U. Nageldinger: An Operating System for Custom Computing Machines based on the Xputer Paradigm; Proceedings of 7th International Workshop on Field Programmable Logic, FPL'97, London, UK, September 1-3, 1997

An Operating System for Custom Computing Machines based on the Xputer Paradigm

Rainer Kress

Siemens AG, Corporate Technology
D-81730 Munich, Germany

Reiner W. Hartenstein, Ulrich Nageldinger

University of Kaiserslautern
D-67663 Kaiserslautern, Germany

Abstract. The paper presents an operating system (OS) for custom computing machines (CCMs) based on the Xputer paradigm. Custom computing tries to combine traditional computing with programmable hardware, attempting to gain from the benefits of both adaptive software and optimized hardware. The OS running as an extension to the actual host OS allows a greater flexibility in deciding what parts of the application should run on the configurable hardware with structural code and what on the host-hardware with conventional software. This decision can be taken late - at run-time - and dynamically, in contrast to early partitioning and deciding at compile-time as used currently on CCMs. Thus the CCM can be used concurrently by multiple users or applications without knowledge of each other. This raises programming and using CCMs to levels close to modern OSes for sequential von Neumann processors.

1 Introduction

With the availability of FPGAs a new class of computing machines has been made possible, the custom computing machines (CCMs). Custom computing tries to combine traditional computing (microprocessor and memory) with programmable hardware, attempting to gain from the benefits of both adaptive software and optimized hardware. A number of different application areas have been proposed for these configurable computing machines. A key characteristic of the applications which reach high performance is the availability of some amount of *semi-static* information. Semi-static means it changes frequently enough to require programmable hardware, but slowly enough to provide the opportunity to improve performance through customized hardware. Three main application areas are [MaSm97]: *situation based* configuration, i.e. configuration changes only at a relatively slow rate; *time-sharing* of hardware resources; and *dynamic circuit generation*, i.e. in evolutionary systems. The latter application uses the full advantage of reconfigurable hardware.

Custom computing machines (CCMs) can be distinguished by their operation mode. Either CCMs run stand-alone or with the aid of a host computer. The use of a host computer starts from simple I/O tasks up to a sophisticated partitioning between tasks scheduled to run on the configurable hardware and the microprocessor of the host. In this paper, we consider host based CCMs. Their programming requires techniques from the hardware/software codesign area [Buch94].

Programming of CCMs can be distinguished into two major groups. The first group contains CCMs, where hardware and software is programmed separately. In most cases the host is programmed using a procedural language like C and the configurable hardware is programmed using schematic entry or synthesized using a hardware description language (HDL) like VHDL. The synchronization is then completely in the responsibility of the user. In the second group, common description languages are used. Either this is a hardware description language (HDL) [GHK90] or a hardware-oriented programming language [SWA93]. These systems require a partitioning step to separate code to be executed on the configurable hardware and on the host.

The new approach proposed in this paper uses also a common description language. But it compiles and synthesizes as much functions as possible for both the programmable hardware and the host. This allows a greater flexibility in deciding what runs on hardware and what on software. The decision can be taken late - at run-time - and dynamically, in contrast to early partitioning and deciding at compile-time. Thus the CCM can be used concurrently by multiple users or applications without knowledge of each other, providing a virtual processor for each user as known from modern operating systems (OS) for von Neumann-processors. The operating system required for such a novel approach is explained using an exemplary CCM based on the Xputer paradigm [HHW90].

The paper is organized as follows: section 2 gives a brief overview on the target architecture. Details can be found in [HaRe95]. Section 3 describes the programming environment CoDeX [HBH96]. The novel approach using an operating system to control the programmable hardware is explained in section 4. Finally the paper is concluded.

2 Overview on the Xputer architecture

Xputer-based CCMs may consist of several Xputer modules. The modules are connected to a host computer. Making use of the host simplifies disk access and all other I/O operations. After setup, each Xputer module runs independently from the others and the host computer until a complete task is processed. Each module generates an interrupt to the host when the task is finished. So, the host is free to concurrently execute other tasks in-between. This allows the use of the Xputer modules as a general purpose acceleration board for time critical parts in an application.

The basic structure of an Xputer module consists of three major parts (figure 1):

- the data memory
- the reconfigurable arithmetic and logic unit (rALU) including several rALU subnets with multiple scan windows (SW)
- the data sequencer (DS)

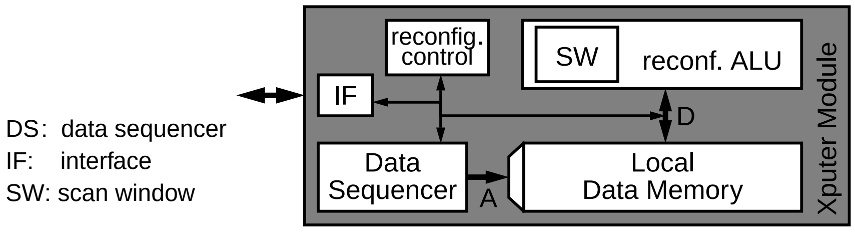

Fig. 1. Xputer module

The data memory contains the data which has to be accessed or modified during an application. The data is arranged in a special order to optimize the data access sequences. This arrangement is called datamap. The data sequencer (DS) computes generic address sequences described by few parameters to access the data. These sequences are called scan patterns. Data sequencing in general means, that the data is addressed in the correct sequence by generic scan patterns and loaded into the rALU. All data manipulations are done in the rALU. It consists of several rALU subnets. After finishing, the rALU signals the end of the computation to the data sequencer.

For the evaluation of standard C programs, word-oriented operators such as addition, subtraction, or multiplication are required. This realization of a reconfigurable architecture for word-oriented operators needs a more coarse grained approach for the logic block architecture of the rALU than supplied by commercially available FPGAs. Therefore, in the current Xputer prototype, a parallel ALU array is used. The ALU array features logic blocks, called DPUs (Datapath Units), which can be loaded with a complete arithmetic or logic operator. The DPUs are capable of executing all operators of the C-language, and additional operators can be added by microprogramming. In the prototype the rALU consists of 96 DPUs. As the regular structure of the parallel ALU array reaches across chip boundaries, the array consisting of several chips can be seen as a large parallel ALU array. Thus, it can be easily adapted to the size of the problem. A more detailed description can be found in [Kres96].

3 The CoDe-X Framework

This section gives an overview on the hardware/software co-design framework CoDe-X (figure 2) [HBH96]. Input language to the framework is the programming language X-C (Xputer C). X-C provides the complete functionality of C with optional extensions (Xputer specific functions) for experienced users. The input is partitioned in a first level into a part for execution on the host and a part for execution on the accelerator. The part for the Xputer-based accelerator is then partitioned in a second level into a sequential part for programming the data sequencer and a structural part for configuring the rALU. Special Xputer-specific functions in the original description of the application allow to take full usage of the high speed-up of the Xputer paradigm. The experienced user may add new functions to a library such that every user can access them.

3.1 Profiling-driven Host/Accelerator Partitioning

The first level of the partitioning process is responsible for the decision which task should be evaluated on the Xputer and which one on the host. Generally three kinds of tasks can be determined:

- tasks which contain dynamic structures, called host tasks,
- Xputer tasks, and
- Xputer-library functions.

The host tasks have to be evaluated on the host, since they cannot be performed on the Xputer accelerator. The Xputer tasks are the candidates for the performance analysis on the host and on the Xputer. The Xputer-library functions are used to get the highest performance out of the Xputer. An experienced user has developed a function library with Xputer-library functions together with their performance values. These functions can be called directly in the input C-programs. The usage of a programmable accelerator arise two problems: the accelerator needs reconfiguration at run-time and in a few cases, the datamap has to be reorganized also at run-time.

3.2 Experienced User Features

The experienced user can use and build optionally a generic function library with a large repertory of Xputer-library functions described in the language MoPL [ABH93]. It fully

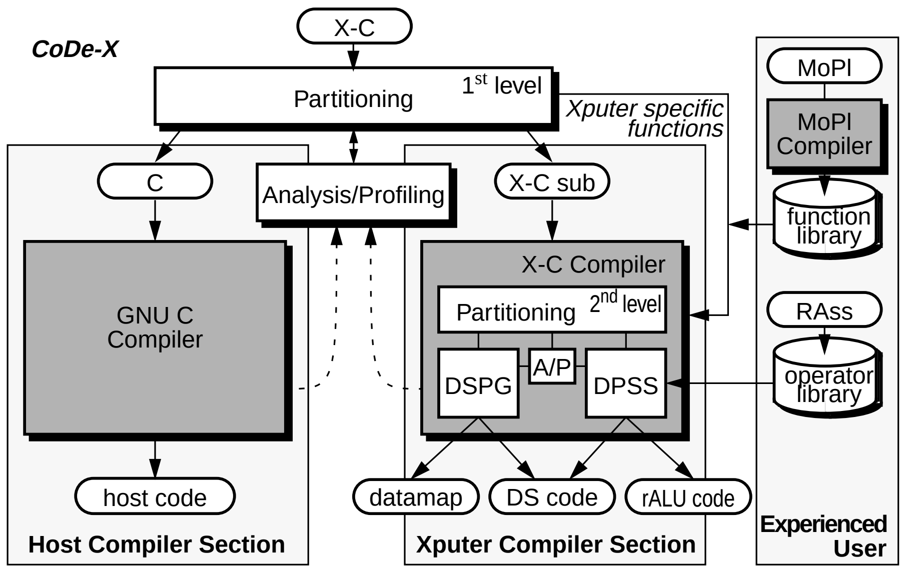

Fig. 2. Overview of the CoDe-X environment

supports all features of the Xputer modules. This optional data-procedural language features make highest acceleration factors provided by Xputer hardware possible.

3.3 Resource-driven second level Partitioning

The X-C compiler [Schm94] translates a program written in a subset of the X-C language into code which can be executed on the Xputer without further user interaction. It comprises four major parts: the data sequencer parameter generator (DSPG), the datapath synthesis system (DPSS), and the analysis/profiling tool (A/P) together with the partitioner. For each part of code delivered from the first partitioning level, the X-C compiler produces a trade-off between area, that means number of DPUs in the rALU and performance.

4 Execution Mechanisms

To enable efficient access to the Xputer hardware, the software interface must provide a way of coupling the Xputer-Tasks to host tasks, which should be conform to the host OS. Thus, the Xputer hardware, namely the rALU, the local memory of the Xputer, the configuration memory of the data sequencer appear as resources to be managed by a software interface.

The current approach for such an interface is the MoM-Runtime-System (MRTS). MoM (Map oriented Machine) is the name of the Xputer prototype. In the MRTS approach, all resources are given temporally to one task. The host-application has to start up the runtime-system, which will take over the Xputer with its modules for the time of its existence in the system, load the configuration and the input data, start an Xputer module, output the results of the calculations and finish (figure 4). This method enables only one task to use the accelerator module at a time, as the runtime-system has to establish a temporary, exclusive control connection to the Xputer, which lasts only during the lifetime of the MRTS. Therefore, there can be only one single process (controlled by a single user) using the Xputer. Thus, a global, static scheduling of all Xputer-tasks of this process is necessary. Furthermore, the temporary control connection mentioned above contains direct accesses to the hardware, including the reception of interrupts from the Xputer.

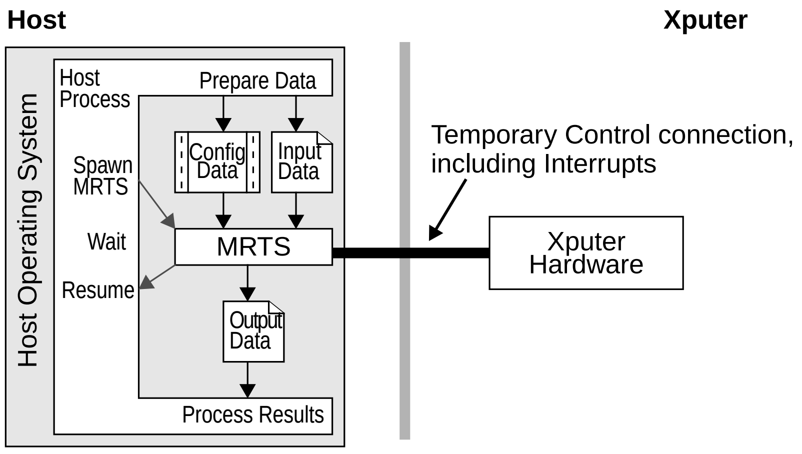

Fig. 3. Primitive Xputer-task execution

4.1 The Xputer Operating System and its executing mechanisms

A new approach has been evolved, which tries to avoid the above restrictions. This approach tries to establish similar powerful resource management, flexibility and efficiency for the Xputer hardware like modern Operating Systems show for standard system resources. Therefore, we call our approach XOS, the Xputer operating system.

The XOS is started only once. It provides access to the Xputer for multiple host tasks. The basic structure of the XOS embedding can be seen in figure 4.

The capabilities of the XOS include dynamic scheduling of tasks and dynamic management of the Xputer memory. Due to the organization of the rALU, the DPUs can also be managed as a resource similar to memory. This enables multiple tasks (configurations) to be present at a time inside the ALU array. The same is true for the configuration memory of the data sequencer.

As the XOS resembles a kind of server in the host-OS, processes can easily communicate and therefore use the Xputer capabilities, making the Xputer a powerful computation resource to traditional applications. The possibilities for a host process to use the Xputer hardware are described in the following.

4.2 Library functions

Library function execution on Xputer modules only. First, a library can contain Xputer specific functions (figure 4 top). This is the most comfortable way for programmers to make use of the Xputer, as there is just one call to the appropriate function, and there is no knowledge about Xputer programming needed. The functions in the libraries have to be generic, that means, that e.g. array boundaries have to be provided by parameters from the side of the host process. These parameters must then be replaced in the

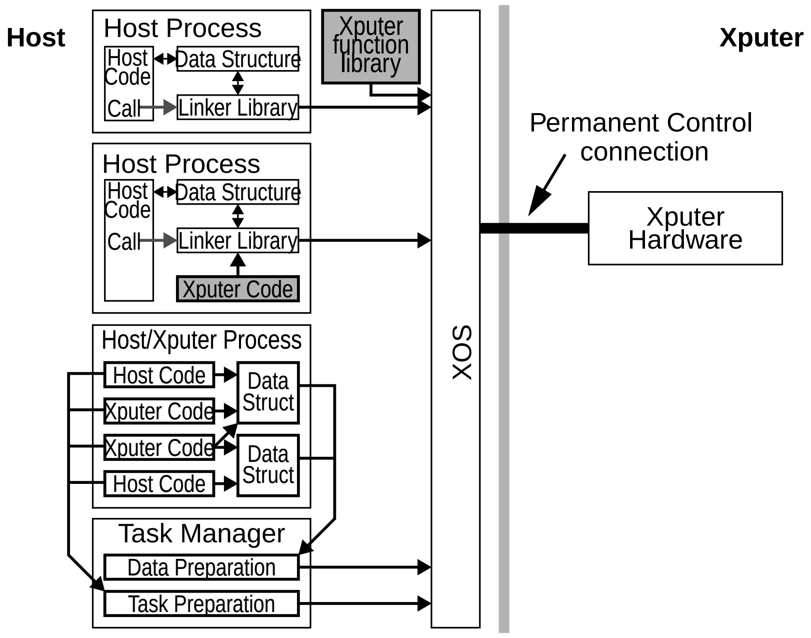

Fig. 4. The XOS approach showing three modes of utilising the Xputer hardware

Xputer function while it is loaded. Therefore, a linker concept is necessary for this mode of operation. This type of Xputer-access is already provided in the CoDe-X framework.

Library function on Xputer modules or the host. Furthermore, a library function may exist not only as an Xputer implementation but also as code executable by the host (figure 4 bottom). Then, the XOS can consider the current load of the Xputer and dynamically select either the host code or the Xputer code to gain the best performance.

Such a scenario is depicted in detail in figure 5. It is assumed, that a host process has an array of integers named „image“ with the dimensions 10 by 10. A filter operation is to be executed on the data in „image“.

To achieve this, the programmer has linked the program with a special linker library. This library contains a function named „XpCall“ in this paper, which initiates the execution of a library function. The parameters for XpCall are:

- The identifier of the requested function („Filter“)
- Pointers to the arrays („image“) containing the data.
- Additional parameters, like array boundaries, filter coefficients etc. For the sake of simplicity, we assume only the array boundaries (10,10) to be passed.

The function XpCall has three main tasks:

1) Contacting the XOS as a client and passing the request, including parameters and array pointers. This data exchange between the XOS and the host process can be established using the interprocess communication (IPC) facilities of the host operating system.

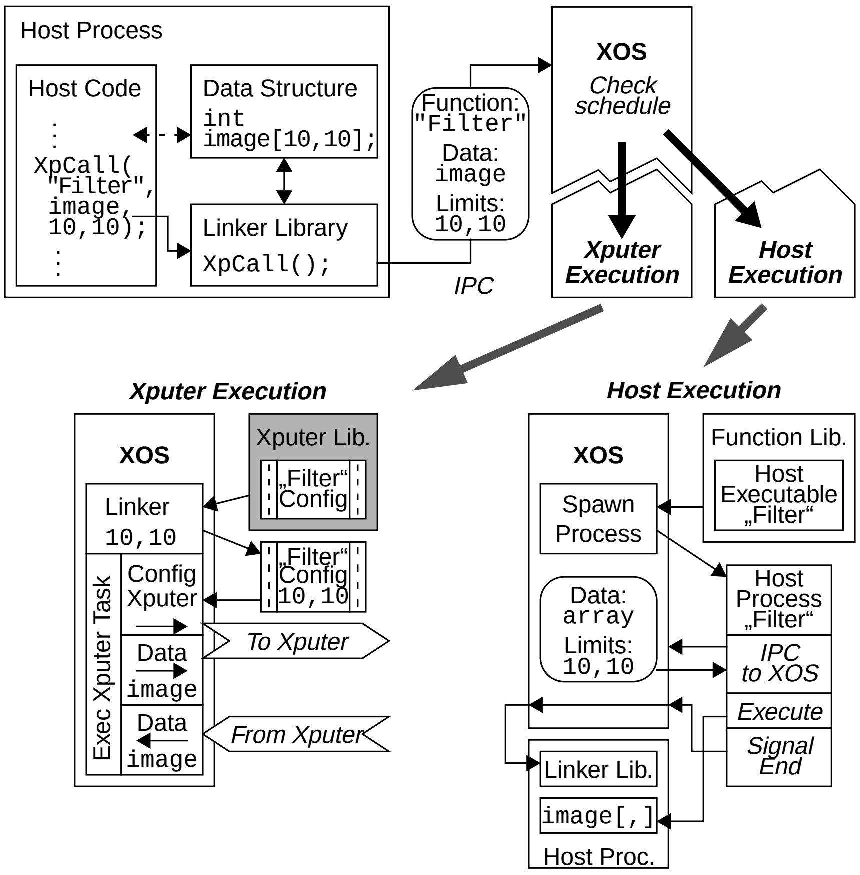

Fig. 5. Example for Library function call

2) Make the data areas (these will normally be arrays) accessible to the XOS or, generally spoken, to a foreign process. On some operating systems, this is a non-trivial task due to sophisticated memory management or security issues. It might be necessary to establish a shared memory segment for the data areas. The Xputer modules directly support shared memories.
3) An obvious requirement for the above two tasks is the knowledge of XpCall about the parameter requirements of the library function. In the example, XpCall must interpret its first argument as an array pointer, but the other two arguments as plain integers. These two variable types need different handling, as the required memory areas to have to be made accessible by foreign processes. Therefore, XpCall knows, that the function „Filter“ needs one array pointer and two plain integers. If the argument list does not match these requirements, an error should be issued. Note that it is still in the responsibility of the programmer, that the first argument is in fact a valid pointer, as XpCall sees only the value of the variable.
There are several ways to let XpCall know what the function „Filter“ needs:

- XpCall simply consults the function library. This would make the linker library, where XpCall resides, unnecessary big, as there must be code included to scan the library and extract the argument list.
- Obtain the information from the XOS, which has to look up the function in the library anyway. This would require a total of three messages between the XOS and the host process: One to pass the function ID to XOS, one to pass the argument list to the process and one to transfer the parameters.
- Encoding the arguments in the function ID, similar to C++ compilers. This is the preferred possibility. In our example, if the function ID is supposed to be a character string, the function „Filter“ could be called „Filter_P1I2“ indicating, that there are one pointer and two integer parameters to be passed. It provides the possibility for generic functions, which have the same name but different argument lists. Consider e.g. a function „Filter_P1I11“, which resembles a filter function taking a pointer, two array boundaries and nine more integers resembling additional coefficients. Then the function „Filter_P1I2“ would be derived from „Filter_P1I11“ by simply providing default values for the nine missing parameters. The default values would be stored with the library function and applied by the XOS.

When the XOS receives the request, it looks up the function in the function library, testing if it exists. The current scheduling decides whether to execute the function on the Xputer or on the host (see also section 5). If the host is chosen, the XOS would create a host process for the function. The lower part of figure 6 shows, how the execution on Xputer or host respectively works.

Xputer. If the function is to be run on the Xputer, the XOS looks up the Xputer configuration in the function library. Normally, this configuration has to be completed with certain parameters. In our example, this would be the array limits (10,10). This procedure of satisfying extern references in the Xputer configuration is similar to a „linking“ process, which is why we call this part of the XOS the linker. After the linking, the configuration is ready to be executed and the XOS will schedule an Xputer task for the function.

Host. If the function is to be run on the host, the XOS looks up the according host executable in the function library. This executable is then spawned as a host process. The new process then contacts the XOS via IPC, receiving the parameters of the function. Having direct access to the data areas of the calling process through the passed pointers, the function can execute and work on the original arrays. When finished, the function signals its termination to the XOS, which passes the signal on to the caller.

4.3 Host processes with Xputer code

Besides using a function provided by a library, an experienced programmer may also create Xputer code directly. To generate the Xputer code, the experienced programmer can use the language MoPL [ABH93].

To execute this code, the program calls a function of a linker library, which handles the communication to the XOS. The resulting configuration code may then be converted to a C data object and linked to the application. As an example, we assume an application, which executes a filter operation onto an internal array, like the example shown in figure 5. Again, this filter should run on the accelerator. Instead of using a library function, the programmer has written an own function in MoPL. The MoPL compiler generates an Xputer configuration for the filter function. In order to make the application consist of one file, the configuration can easily be transformed into a C source file by creating an array containing the configuration binary. The source file can then be compiled and linked together with the application source code. To enable Xputer access for the application, the programmer has again linked the code with the linker library, which coordinates communication to the XOS.

At run time, the application contains the Xputer code as a normal data object. When the filter is to be executed, a function from the linker library named „XpLoad“ is called. The arguments to this function are a pointer to the array with the Xputer configuration and a pointer to the data area. Note, that the array limits are included in the Xputer code, which has been designed by the programmer to match the application requirements. The function „XpLoad“ does basically the same as „XpCall“, which is used for functions from the Xputer library in the example in section 4.2. The difference lies in the simpler handling of the accelerator code, which is transferred directly from the C-array instead of the library. Furthermore, the function will be executed on the Xputer in any way. There is no option for the XOS to use a traditional implementation.

5 Dynamic task scheduling

To provide an efficient use of the Xputer hardware, a good scheduling of the Xputer tasks has to be performed. As the accelerator should be a global resource to the system, the ability of multitasking is desirable. In this section, the requirements for a task scheduling are discussed.

For the processing of interrupts, the XOS should provide preemptiveness. That means the XOS should be capable to suspend any running task at any time. To preserve the status of the old task, a context switch must be performed. Therefore, the current status of the hardware must be saveable and loadable. In modern von Neumann processors, the whole processor status is contained in a set of registers. Normally, there is a special instruction, which saves or loads the complete register set to or from the system stack, i.e. a location in the main memory. The status of an Xputer module consists of much more data:

- the current status of the data sequencer, consisting of the current configuration as well as the current working-location,
- the status of the rALU, which comprises the current configuration (ranges from 20 to 100 kbits for 96 DPUs), as well as the status of every used register in each DPU.

It is easily seen, that the status of the Xputer contains more data than that of a traditional processor. Therefore, both data sequencer and rALU provide the possibility to hold multiple configurations in their configuration memories. The context switch is performed as a switch between two configurations. We are aware, that the maximal number of configurations in the hardware limits the capabilities of the XOS, but a transfer of the configuration data to and from an external memory would need too much time.

For the dynamic scheduling, information from the static scheduling of CoDe-X is used. For each task, its predecessor, successor and mobility is stored. From the profiler of the CoDe-X framework, the acceleration factor, required configuration time, and the time for memory access (if not in a local memory) is saved. A cost function (1) using this information creates a priority for the currently available tasks ready for execution.

$$\text{cost function} = f(\text{acc. factor}, t_{\text{config}}, t_{\text{memory}}, \text{mobility}) \tag{1}$$

Running only a single application concurrently on the Xputer results in the same schedule as provided by the CoDe-X environment.

6 Conclusions

An operating system (OS) for custom computing machines (CCMs) based on the Xputer paradigm has been presented. The OS running as extension to the actual host OS allows a greater flexibility in deciding what runs on the configurable hardware and what on the host hardware. This decision is taken late, at run-time and dynamically, in contrast to early partitioning and deciding at compile-time as used currently by CCMs. Thus a CCM can be used concurrently by multiple users or applications without knowledge of each other. Compared to a previous runtime-system, the XOS approach is its logical extension, narrowing the gap to capabilities of modern operating systems for von Neumann computers.

References

[ABH93] A. Ast, J. Becker, R. Hartenstein, R. Kress, H. Reinig, K. Schmidt: MoPL-3: A New High Level Xputer Programming Language; 3rd Int´ Workshop On Field Programmable Logic And Applications, Oxford, 7. - 10. September 1993

[Buch94] Klaus Buchenrieder: Hardware/Software Co-Design, An Annotated Bibliography; IT Press Chicago, 1994

[GHK90] M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minich, P. Olsen: SPLASH: A Reconfigurable Linear Logic Array; International Conference on Parallel Processing, pp I 526 - I 532, 1990

[HBH96] R. W. Hartenstein, J. Becker, M. Herz, R. Kress, U. Nageldinger: A Parallelizing Programming Environment for Embedded Xputer-based Accelerators; High Performance Computing Symposium ‘96, Ottawa, Canada, 1996

[HaRe95] R. W. Hartenstein, H. Reinig: Novel Sequencer Hardware for High-Speed Signal Processing; Workshop on Design Methodologies for Microelectronics, Smolenice Castle, Slovakia, Sept. 11-13, 1995

[HHW90] R. W. Hartenstein, A. G. Hirschbiel, M. Weber: A Novel Paradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware; InfoJapan‘90 - International Conference memorizing the 30th Anniversary of the Computer Society of Japan, Tokyo, Japan, 1990

[Kres96] R. Kress: A fast reconfigurable ALU for Xputers; Ph. D. dissertation, Kaiserslautern University, 1996

[MaSm97] W. Mangione-Smith: Configurable Computing: Concepts and Issues; Task Force on Configurable Computing, Proceeding of the HICSS 97, pp. 710-712, 1997

[Schm94] K. Schmidt: A Program Partitioning, Restructuring, and Mapping Method for Xputers; Ph.D. Thesis, University of Kaiserslautern, 1994

[SWA93] A. Smith, M. Wazlowski, L. Agarwal, T. Lee, E. Lam, P. Athanas, H. Silverman, S. Ghosh: PRISM-II: Compiler and Architecture; IEEE Workshop on FPGAs for Custom Computing Machines, FCCM‘93, Napa, CA, April 1993

R. W. Hartenstein, R. Kress, U. Nageldinger: An Operating System for Custom Computing Machines based on the Xputer Paradigm; Proceedings of 7th International Workshop on Field Programmable Logic, FPL'97, London, UK, September 1-3, 1997

An Operating System for Custom Computing Machines based on the Xputer Paradigm

Rainer Kress
Siemens AG, Corporate Technology
D-81730 Munich, Germany

Reiner W. Hartenstein, Ulrich Nageldinger
University of Kaiserslautern
D-67663 Kaiserslautern, Germany

Abstract. The paper presents an operating system (OS) for custom computing machines (CCMs) based on the Xputer paradigm. Custom computing tries to combine traditional computing with programmable hardware, attempting to gain from the benefits of both adaptive software and optimized hardware. The OS running as an extension to the actual host OS allows a greater flexibility in deciding what parts of the application should run on the configurable hardware with structural code and what on the host-hardware with conventional software. This decision can be taken late - at run-time - and dynamically, in contrast to early partitioning and deciding at compile-time as used currently on CCMs. Thus the CCM can be used concurrently by multiple users or applications without knowledge of each other. This raises programming and using CCMs to levels close to modern OSes for sequential von Neumann processors.

1 Introduction

With the availability of FPGAs a new class of computing machines has been made possible, the custom computing machines (CCMs). Custom computing tries to combine traditional computing (microprocessor and memory) with programmable hardware, attempting to gain from the benefits of both adaptive software and optimized hardware. A number of different application areas have been proposed for these configurable computing machines. A key characteristic of the applications which reach high performance is the availability of some amount of *semi-static* information. Semi-static means it changes frequently enough to require programmable hardware, but slowly enough to provide the opportunity to improve performance through customized hardware. Three main application areas are [MaSm97]: *situation based* configuration, i.e. configuration changes only at a relatively slow rate; *time-sharing* of hardware resources; and *dynamic circuit generation*, i.e. in evolutionary systems. The latter application uses the full advantage of reconfigurable hardware.

Custom computing machines (CCMs) can be distinguished by their operation mode. Either CCMs run stand-alone or with the aid of a host computer. The use of a host computer starts from simple I/O tasks up to a sophisticated partitioning between tasks scheduled to run on the configurable hardware and the microprocessor of the host. In this paper, we consider host based CCMs. Their programming requires techniques from the hardware/software codesign area [Buch94].

Programming of CCMs can be distinguished into two major groups. The first group contains CCMs, where hardware and software is programmed separately. In most cases the host is programmed using a procedural language like C and the configurable hardware is programmed using schematic entry or synthesized using a hardware description language (HDL) like VHDL. The synchronization is then completely in the responsibility of the user. In the second group, common description languages are used. Either this is a hardware description language (HDL) [GHK90] or a hardware-oriented programming language [SWA93]. These systems require a partitioning step to separate code to be executed on the configurable hardware and on the host.

The new approach proposed in this paper uses also a common description language. But it compiles and synthesizes as much functions as possible for both the programmable hardware and the host. This allows a greater flexibility in deciding what runs on hardware and what on software. The decision can be taken late - at run-time - and dynamically, in contrast to early partitioning and deciding at compile-time. Thus the CCM can be used concurrently by multiple users or applications without knowledge of each other, providing a virtual processor for each user as known from modern operating systems (OS) for von Neumann-processors. The operating system required for such a novel approach is explained using an exemplary CCM based on the Xputer paradigm [HHW90].

The paper is organized as follows: section 2 gives a brief overview on the target architecture. Details can be found in [HaRe95]. Section 3 describes the programming environment CoDeX [HBH96]. The novel approach using an operating system to control the programmable hardware is explained in section 4. Finally the paper is concluded.

2 Overview on the Xputer architecture

Xputer-based CCMs may consist of several Xputer modules. The modules are connected to a host computer. Making use of the host simplifies disk access and all other I/O operations. After setup, each Xputer module runs independently from the others and the host computer until a complete task is processed. Each module generates an interrupt to the host when the task is finished. So, the host is free to concurrently execute other tasks in-between. This allows the use of the Xputer modules as a general purpose acceleration board for time critical parts in an application.

The basic structure of an Xputer module consists of three major parts (figure 1):

- the data memory
- the reconfigurable arithmetic and logic unit (rALU) including several rALU subnets with multiple scan windows (SW)
- the data sequencer (DS)

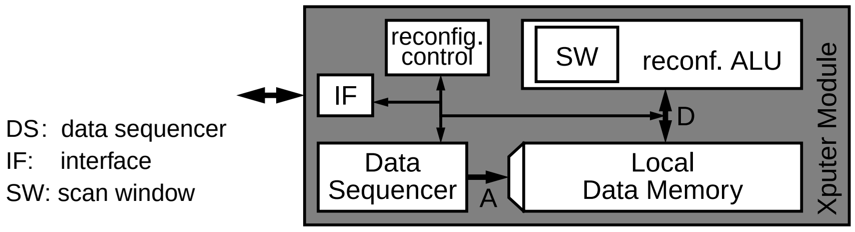

Fig. 1. Xputer module

The data memory contains the data which has to be accessed or modified during an application. The data is arranged in a special order to optimize the data access sequences. This arrangement is called datamap. The data sequencer (DS) computes generic address sequences described by few parameters to access the data. These sequences are called scan patterns. Data sequencing in general means, that the data is addressed in the correct sequence by generic scan patterns and loaded into the rALU. All data manipulations are done in the rALU. It consists of several rALU subnets. After finishing, the rALU signals the end of the computation to the data sequencer.

For the evaluation of standard C programs, word-oriented operators such as addition, subtraction, or multiplication are required. This realization of a reconfigurable architecture for word-oriented operators needs a more coarse grained approach for the logic block architecture of the rALU than supplied by commercially available FPGAs. Therefore, in the current Xputer prototype, a parallel ALU array is used. The ALU array features logic blocks, called DPUs (Datapath Units), which can be loaded with a complete arithmetic or logic operator. The DPUs are capable of executing all operators of the C-language, and additional operators can be added by microprogramming. In the prototype the rALU consists of 96 DPUs. As the regular structure of the parallel ALU array reaches across chip boundaries, the array consisting of several chips can be seen as a large parallel ALU array. Thus, it can be easily adapted to the size of the problem. A more detailed description can be found in [Kres96].

3 The CoDe-X Framework

This section gives an overview on the hardware/software co-design framework CoDe-X (figure 2) [HBH96]. Input language to the framework is the programming language X-C (Xputer C). X-C provides the complete functionality of C with optional extensions (Xputer specific functions) for experienced users. The input is partitioned in a first level into a part for execution on the host and a part for execution on the accelerator. The part for the Xputer-based accelerator is then partitioned in a second level into a sequential part for programming the data sequencer and a structural part for configuring the rALU. Special Xputer-specific functions in the original description of the application allow to take full usage of the high speed-up of the Xputer paradigm. The experienced user may add new functions to a library such that every user can access them.

3.1 Profiling-driven Host/Accelerator Partitioning

The first level of the partitioning process is responsible for the decision which task should be evaluated on the Xputer and which one on the host. Generally three kinds of tasks can be determined:

- tasks which contain dynamic structures, called host tasks,
- Xputer tasks, and
- Xputer-library functions.

The host tasks have to be evaluated on the host, since they cannot be performed on the Xputer accelerator. The Xputer tasks are the candidates for the performance analysis on the host and on the Xputer. The Xputer-library functions are used to get the highest performance out of the Xputer. An experienced user has developed a function library with Xputer-library functions together with their performance values. These functions can be called directly in the input C-programs. The usage of a programmable accelerator arise two problems: the accelerator needs reconfiguration at run-time and in a few cases, the datamap has to be reorganized also at run-time.

3.2 Experienced User Features

The experienced user can use and build optionally a generic function library with a large repertory of Xputer-library functions described in the language MoPL [ABH93]. It fully

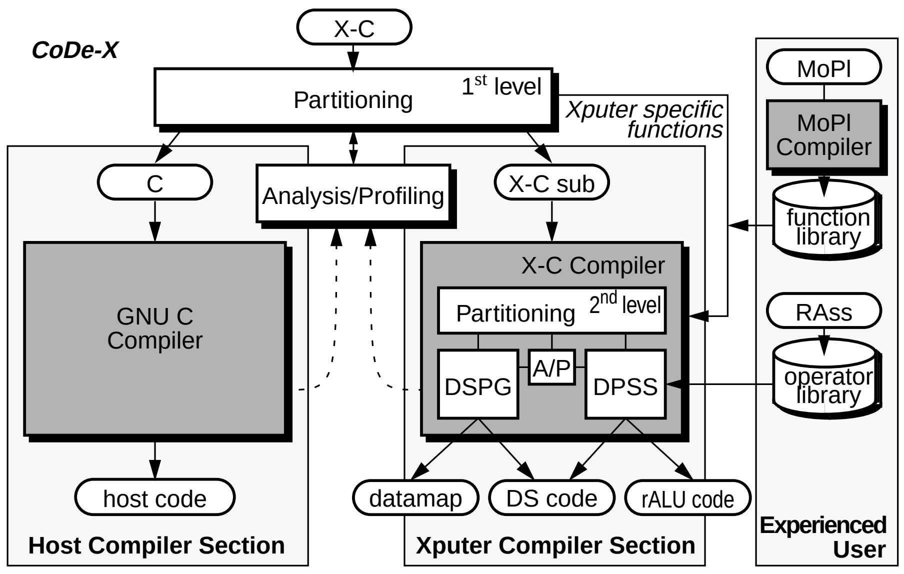

Fig. 2. Overview of the CoDe-X environment

supports all features of the Xputer modules. This optional data-procedural language features make highest acceleration factors provided by Xputer hardware possible.

3.3 Resource-driven second level Partitioning

The X-C compiler [Schm94] translates a program written in a subset of the X-C language into code which can be executed on the Xputer without further user interaction. It comprises four major parts: the data sequencer parameter generator (DSPG), the datapath synthesis system (DPSS), and the analysis/profiling tool (A/P) together with the partitioner. For each part of code delivered from the first partitioning level, the X-C compiler produces a trade-off between area, that means number of DPUs in the rALU and performance.

4 Execution Mechanisms

To enable efficient access to the Xputer hardware, the software interface must provide a way of coupling the Xputer-Tasks to host tasks, which should be conform to the host OS. Thus, the Xputer hardware, namely the rALU, the local memory of the Xputer, the configuration memory of the data sequencer appear as resources to be managed by a software interface.

The current approach for such an interface is the MoM-Runtime-System (MRTS). MoM (Map oriented Machine) is the name of the Xputer prototype. In the MRTS approach, all resources are given temporally to one task. The host-application has to start up the runtime-system, which will take over the Xputer with its modules for the time of its existence in the system, load the configuration and the input data, start an Xputer module, output the results of the calculations and finish (figure 4). This method enables only one task to use the accelerator module at a time, as the runtime-system has to establish a temporary, exclusive control connection to the Xputer, which lasts only during the lifetime of the MRTS. Therefore, there can be only one single process (controlled by a single user) using the Xputer. Thus, a global, static scheduling of all Xputer-tasks of this process is necessary. Furthermore, the temporary control connection mentioned above contains direct accesses to the hardware, including the reception of interrupts from the Xputer.

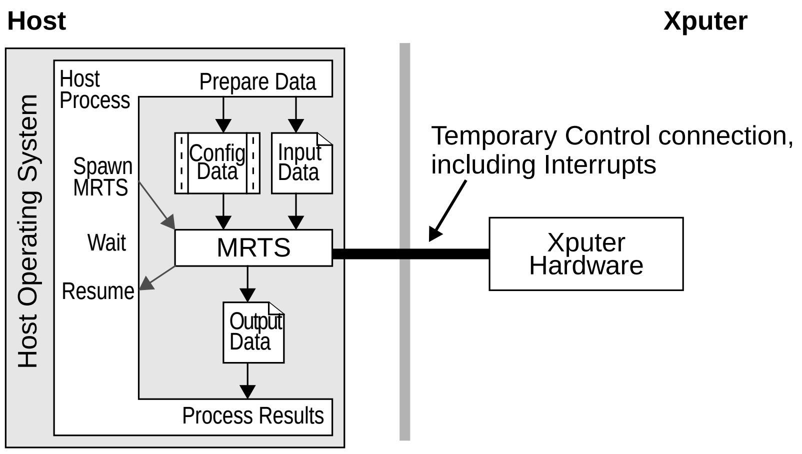

Fig. 3. Primitive Xputer-task execution

4.1 The Xputer Operating System and its executing mechanisms

A new approach has been evolved, which tries to avoid the above restrictions. This approach tries to establish similar powerful resource management, flexibility and efficiency for the Xputer hardware like modern Operating Systems show for standard system resources. Therefore, we call our approach XOS, the Xputer operating system.

The XOS is started only once. It provides access to the Xputer for multiple host tasks. The basic structure of the XOS embedding can be seen in figure 4.

The capabilities of the XOS include dynamic scheduling of tasks and dynamic management of the Xputer memory. Due to the organization of the rALU, the DPUs can also be managed as a resource similar to memory. This enables multiple tasks (configurations) to be present at a time inside the ALU array. The same is true for the configuration memory of the data sequencer.

As the XOS resembles a kind of server in the host-OS, processes can easily communicate and therefore use the Xputer capabilities, making the Xputer a powerful computation resource to traditional applications. The possibilities for a host process to use the Xputer hardware are described in the following.

4.2 Library functions

Library function execution on Xputer modules only. First, a library can contain Xputer specific functions (figure 4 top). This is the most comfortable way for programmers to make use of the Xputer, as there is just one call to the appropriate function, and there is no knowledge about Xputer programming needed. The functions in the libraries have to be generic, that means, that e.g. array boundaries have to be provided by parameters from the side of the host process. These parameters must then be replaced in the

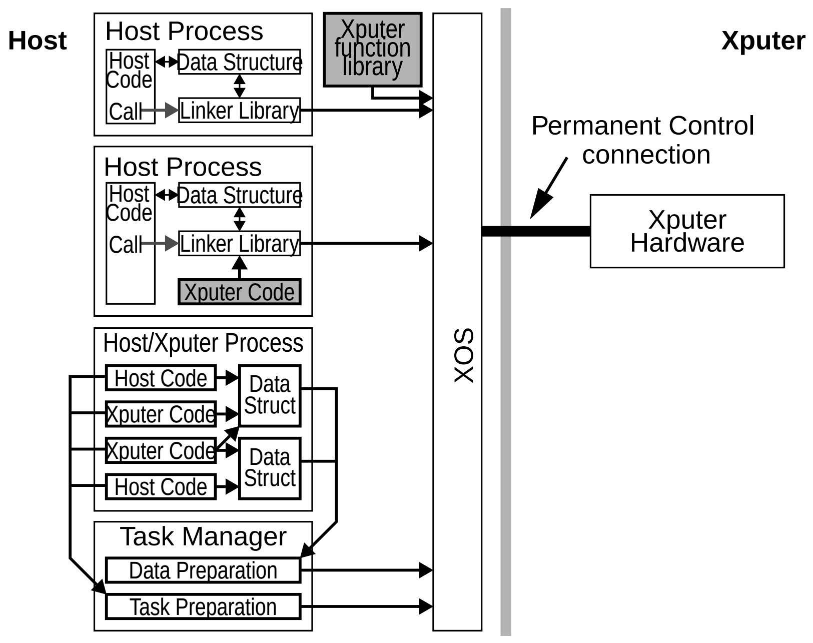

Fig. 4. The XOS approach showing three modes of utilising the Xputer hardware

Xputer function while it is loaded. Therefore, a linker concept is necessary for this mode of operation. This type of Xputer-access is already provided in the CoDe-X framework.

Library function on Xputer modules or the host. Furthermore, a library function may exist not only as an Xputer implementation but also as code executable by the host (figure 4 bottom). Then, the XOS can consider the current load of the Xputer and dynamically select either the host code or the Xputer code to gain the best performance.

Such a scenario is depicted in detail in figure 5. It is assumed, that a host process has an array of integers named „image“ with the dimensions 10 by 10. A filter operation is to be executed on the data in „image“.

To achieve this, the programmer has linked the program with a special linker library. This library contains a function named „XpCall“ in this paper, which initiates the execution of a library function. The parameters for XpCall are:

- The identifier of the requested function („Filter“)
- Pointers to the arrays („image“) containing the data.
- Additional parameters, like array boundaries, filter coefficients etc. For the sake of simplicity, we assume only the array boundaries (10,10) to be passed.

The function XpCall has three main tasks:

1) Contacting the XOS as a client and passing the request, including parameters and array pointers. This data exchange between the XOS and the host process can be established using the interprocess communication (IPC) facilities of the host operating system.

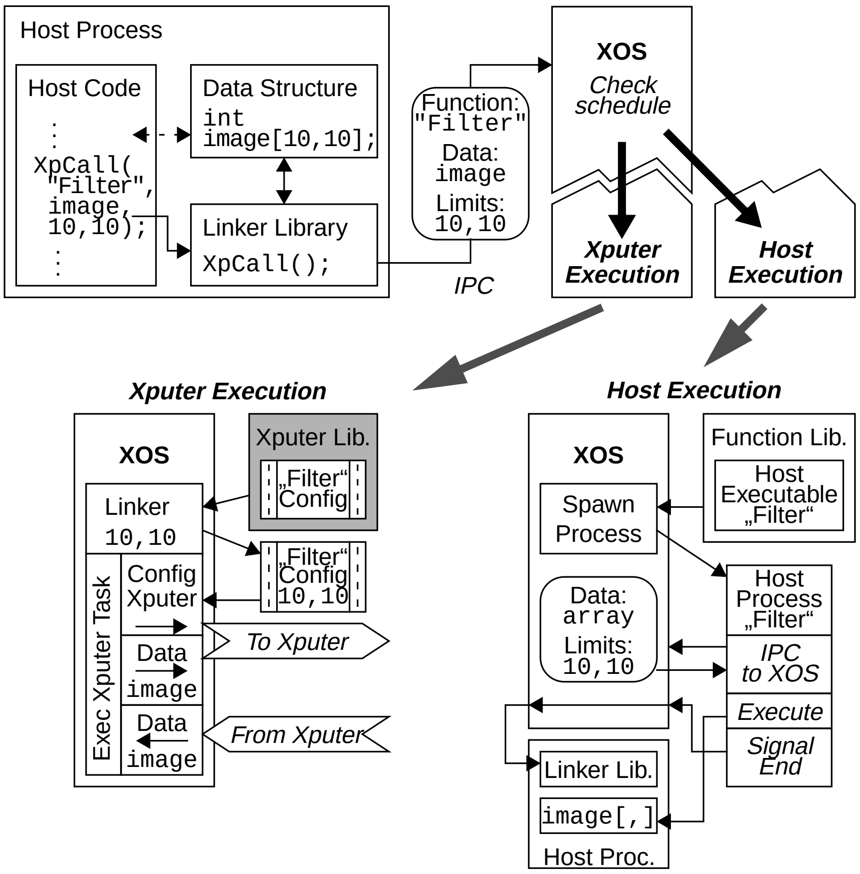

Fig. 5. Example for Library function call

2) Make the data areas (these will normally be arrays) accessible to the XOS or, generally spoken, to a foreign process. On some operating systems, this is a non-trivial task due to sophisticated memory management or security issues. It might be necessary to establish a shared memory segment for the data areas. The Xputer modules directly support shared memories.
3) An obvious requirement for the above two tasks is the knowledge of XpCall about the parameter requirements of the library function. In the example, XpCall must interpret its first argument as an array pointer, but the other two arguments as plain integers. These two variable types need different handling, as the required memory areas to have to be made accessible by foreign processes. Therefore, XpCall knows, that the function „Filter“ needs one array pointer and two plain integers. If the argument list does not match these requirements, an error should be issued. Note that it is still in the responsibility of the programmer, that the first argument is in fact a valid pointer, as XpCall sees only the value of the variable.
There are several ways to let XpCall know what the function „Filter“ needs:

- XpCall simply consults the function library. This would make the linker library, where XpCall resides, unnecessary big, as there must be code included to scan the library and extract the argument list.
- Obtain the information from the XOS, which has to look up the function in the library anyway. This would require a total of three messages between the XOS and the host process: One to pass the function ID to XOS, one to pass the argument list to the process and one to transfer the parameters.
- Encoding the arguments in the function ID, similar to C++ compilers. This is the preferred possibility. In our example, if the function ID is supposed to be a character string, the function „Filter" could be called „Filter_P1I2" indicating, that there are one pointer and two integer parameters to be passed. It provides the possibility for generic functions, which have the same name but different argument lists. Consider e.g. a function „Filter_P1I11", which resembles a filter function taking a pointer, two array boundaries and nine more integers resembling additional coefficients. Then the function „Filter_P1I2" would be derived from „Filter_P1I11" by simply providing default values for the nine missing parameters. The default values would be stored with the library function and applied by the XOS.

When the XOS receives the request, it looks up the function in the function library, testing if it exists. The current scheduling decides whether to execute the function on the Xputer or on the host (see also section 5). If the host is chosen, the XOS would create a host process for the function. The lower part of figure 6 shows, how the execution on Xputer or host respectively works.

Xputer. If the function is to be run on the Xputer, the XOS looks up the Xputer configuration in the function library. Normally, this configuration has to be completed with certain parameters. In our example, this would be the array limits (10,10). This procedure of satisfying extern references in the Xputer configuration is similar to a „linking" process, which is why we call this part of the XOS the linker. After the linking, the configuration is ready to be executed and the XOS will schedule an Xputer task for the function.

Host. If the function is to be run on the host, the XOS looks up the according host executable in the function library. This executable is then spawned as a host process. The new process then contacts the XOS via IPC, receiving the parameters of the function. Having direct access to the data areas of the calling process through the passed pointers, the function can execute and work on the original arrays. When finished, the function signals its termination to the XOS, which passes the signal on to the caller.

4.3 Host processes with Xputer code

Besides using a function provided by a library, an experienced programmer may also create Xputer code directly. To generate the Xputer code, the experienced programmer can use the language MoPL [ABH93].

To execute this code, the program calls a function of a linker library, which handles the communication to the XOS. The resulting configuration code may then be converted to a C data object and linked to the application. As an example, we assume an application, which executes a filter operation onto an internal array, like the example shown in figure 5. Again, this filter should run on the accelerator. Instead of using a library function, the programmer has written an own function in MoPL. The MoPL compiler generates an Xputer configuration for the filter function. In order to make the application consist of one file, the configuration can easily be transformed into a C source file by creating an array containing the configuration binary. The source file can then be compiled and linked together with the application source code. To enable Xputer access for the application, the programmer has again linked the code with the linker library, which coordinates communication to the XOS.

At run time, the application contains the Xputer code as a normal data object. When the filter is to be executed, a function from the linker library named „XpLoad“ is called. The arguments to this function are a pointer to the array with the Xputer configuration and a pointer to the data area. Note, that the array limits are included in the Xputer code, which has been designed by the programmer to match the application requirements. The function „XpLoad“ does basically the same as „XpCall“, which is used for functions from the Xputer library in the example in section 4.2. The difference lies in the simpler handling of the accelerator code, which is transferred directly from the C-array instead of the library. Furthermore, the function will be executed on the Xputer in any way. There is no option for the XOS to use a traditional implementation.

5 Dynamic task scheduling

To provide an efficient use of the Xputer hardware, a good scheduling of the Xputer tasks has to be performed. As the accelerator should be a global resource to the system, the ability of multitasking is desirable. In this section, the requirements for a task scheduling are discussed.

For the processing of interrupts, the XOS should provide preemptiveness. That means the XOS should be capable to suspend any running task at any time. To preserve the status of the old task, a context switch must be performed. Therefore, the current status of the hardware must be saveable and loadable. In modern von Neumann processors, the whole processor status is contained in a set of registers. Normally, there is a special instruction, which saves or loads the complete register set to or from the system stack, i.e. a location in the main memory. The status of an Xputer module consists of much more data:

- the current status of the data sequencer, consisting of the current configuration as well as the current working-location,
- the status of the rALU, which comprises the current configuration (ranges from 20 to 100 kbits for 96 DPUs), as well as the status of every used register in each DPU.

It is easily seen, that the status of the Xputer contains more data than that of a traditional processor. Therefore, both data sequencer and rALU provide the possibility to hold multiple configurations in their configuration memories. The context switch is performed as a switch between two configurations. We are aware, that the maximal number of configurations in the hardware limits the capabilities of the XOS, but a transfer of the configuration data to and from an external memory would need too much time.

For the dynamic scheduling, information from the static scheduling of CoDe-X is used. For each task, its predecessor, successor and mobility is stored. From the profiler of the CoDe-X framework, the acceleration factor, required configuration time, and the time for memory access (if not in a local memory) is saved. A cost function (1) using this information creates a priority for the currently available tasks ready for execution.

$$\text{cost function} = f(\text{acc. factor}, t_{\text{config}}, t_{\text{memory}}, \text{mobility}) \tag{1}$$

Running only a single application concurrently on the Xputer results in the same schedule as provided by the CoDe-X environment.

6 Conclusions

An operating system (OS) for custom computing machines (CCMs) based on the Xputer paradigm has been presented. The OS running as extension to the actual host OS allows a greater flexibility in deciding what runs on the configurable hardware and what on the host hardware. This decision is taken late, at run-time and dynamically, in contrast to early partitioning and deciding at compile-time as used currently by CCMs. Thus a CCM can be used concurrently by multiple users or applications without knowledge of each other. Compared to a previous runtime-system, the XOS approach is its logical extension, narrowing the gap to capabilities of modern operating systems for von Neumann computers.

References

[ABH93] A. Ast, J. Becker, R. Hartenstein, R. Kress, H. Reinig, K. Schmidt: MoPL-3: A New High Level Xputer Programming Language; 3rd Int´ Workshop On Field Programmable Logic And Applications, Oxford, 7. - 10. September 1993

[Buch94] Klaus Buchenrieder: Hardware/Software Co-Design, An Annotated Bibliography; IT Press Chicago, 1994

[GHK90] M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minich, P. Olsen: SPLASH: A Reconfigurable Linear Logic Array; International Conference on Parallel Processing, pp I 526 - I 532, 1990

[HBH96] R. W. Hartenstein, J. Becker, M. Herz, R. Kress, U. Nageldinger: A Parallelizing Programming Environment for Embedded Xputer-based Accelerators; High Performance Computing Symposium ‘96, Ottawa, Canada, 1996

[HaRe95] R. W. Hartenstein, H. Reinig: Novel Sequencer Hardware for High-Speed Signal Processing; Workshop on Design Methodologies for Microelectronics, Smolenice Castle, Slovakia, Sept. 11-13, 1995

[HHW90] R. W. Hartenstein, A. G. Hirschbiel, M. Weber: A Novel Paradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware; InfoJapan‘90 - International Conference memorizing the 30th Anniversary of the Computer Society of Japan, Tokyo, Japan, 1990

[Kres96] R. Kress: A fast reconfigurable ALU for Xputers; Ph. D. dissertation, Kaiserslautern University, 1996

[MaSm97] W. Mangione-Smith: Configurable Computing: Concepts and Issues; Task Force on Configurable Computing, Proceeding of the HICSS 97, pp. 710-712, 1997

[Schm94] K. Schmidt: A Program Partitioning, Restructuring, and Mapping Method for Xputers; Ph.D. Thesis, University of Kaiserslautern, 1994

[SWA93] A. Smith, M. Wazlowski, L. Agarwal, T. Lee, E. Lam, P. Athanas, H. Silverman, S. Ghosh: PRISM-II: Compiler and Architecture; IEEE Workshop on FPGAs for Custom Computing Machines, FCCM‘93, Napa, CA, April 1993