Reiner W. Hartenstein, Juergen Becker, Michael Herz, Ulrich Nageldinger: Data Scheduling in Hardware/Software Co-Design for Field-programmable Accelerators; Proceedings of 7th International Workshop on Field Programmable Logic, FPL'97, London, UK, September 1-3, 1997

Data Scheduling in Hardware/Software Co-Design for field-programmable Accelerators

R. W. Hartenstein, J. Becker, M. Herz, U. Nageldinger
University of Kaiserslautern
Erwin-Schrödinger-Straße, D-67663 Kaiserslautern, Germany
Fax: ++49 631 205 2640, email: abakus@informatik.uni-kl.de
www: http://xputers.informatik.uni-kl.de/

Draft Paper

Abstract

The paper presents a general data and task scheduling technique for parallel reconfigurable accelerators with one or more processing modules and the capability for local and shared memory access. Multiple tasks and their data are mapped by a two-level hardware/software co-design method onto the processing modules and the host, providing a high degree of parallelism. The system performance is enhanced by avoiding unnecessary data transfers. It is shown, how overall system performance can be increased by appropriate data distribution.

1. Introduction

Due to the increasing demands in computation power, it has shown, that there are certain applications, which take too much time to compute on a standard von Neumann Computer, despite of remarkable developments in system speed. Such algorithms can be found e.g. in image or digital signal processing, multimedia applications and others.

CCMs try to boost the performance by adding an external accelerator to the host computer. They are connected to the host via a coprocessor bus or an I/O bus. In the following short discussion of CCMs, we follow the classification given in [HBK96]. A subclass of CCMs uses field-programmable logic for the accelerator. Therefore, these CCMs are called F-CCMs. Field-programmable logic provides the advantage of a manifold of possible configurations of the accelerator, which is therefore not restricted to execution of one dedicated application. In the following, only this class of accelerators is considered. These F-CCMs will be classified in the following by the way the data memory is attached and accessed.

Typically, four classes of memory accessing-methods can be distinguished (see figure 1):

a) Remote memory access: The accelerator uses the host memory, which is accessed through the host CPU. This means, the accelerator resembles a kind of coprocessor, like a floating-point processor. In contrast to coprocessors, the

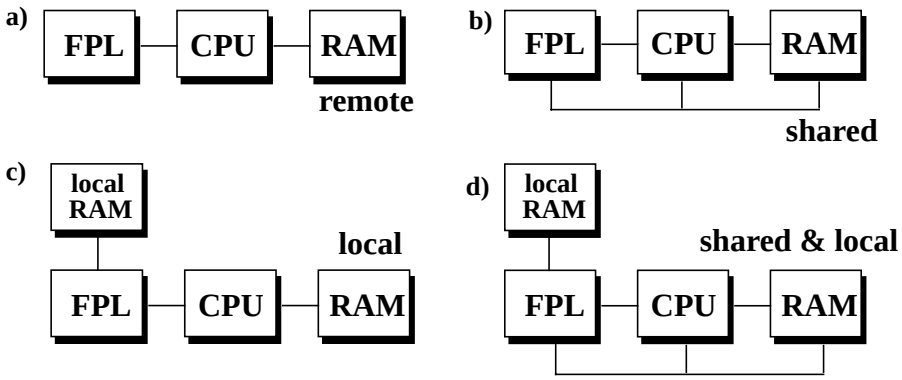

Figure 1. Memory organization of F-CCMs

accelerator benefits from the field-programmable logic, making it more flexible than a dedicated hardware. Examples for F-CCMs with remote memory access are the PRISM and PRISM-II [Ath92] [AtSi93] [SWA93].

b) Shared memory access: The accelerator uses also the host memory, but has an own channel to access it, concurrently to the host CPU. This gives the accelerator the possibility to run independently and parallel to the CPU. An example for this type is the general-purpose F-CCM ArMen [RLR93].

c) Local memory: Accelerators, which have local memory need not access the host memory concurrently to the CPU during execution. We assume for this classification, that there is also no possibility for the accelerator to access the host memory directly. So, this constellation needs a transfer of the application data by the host CPU to the local memory in the configuration phase. A performance gain is visible, if the execution time on the accelerator is long compared to the configuration time and the time to transfer the data. Examples for this class are the commercial F-CCM EVC [CCA95] and Enable++ [EVC94].

d) Local memory and shared memory access: Here, the Accelerator has both local memory and access to the host memory. The local memory enables the accelerator to calculate independently from the host, avoding concurrent requests for the bus, which slow down the system. Additionally, necessary accesses to the host memory for transfering application data are feasible without charging the CPU with additional load. This approach requires a more sophisticated hardware than the other ones. Also, to gain an increase of performance, the data has to be distributed in an adequate way, so that overheads due to data transfers between host and accelerator are minimized. Examples for this type of memory communication are SPLASH [GHK90] [GHK91], PeRLe-1 [BRV93] [BeTe94].

Our own approach follows the Xputer paradigm [AHR94] [HHS91] [Hirs91]. The current prototype is called MoM-3 (Map oriented Machine 3) [BHK95] . The MoM-3

uses multiple Xputer modules, which have local memories as well as access to the host memory. It has been seen in the comparison above, that the shared and local memory solution involves the potential to gain a remarkable increase of performance. Data, which is accessed frequently, is copied to the local memory. Data, which is accessed only once remains on the host. Therefore, it exists the necessity for an effective distribution of the data onto the different memory resources at compile time, which is called data scheduling. This paper will focus on the problems and appropriate solution strategies relevant to these issues. In our current Application Development Environment, the two-level hardware/software co-design framework Code-X (Co-Design for Xputers) [HaBe97a], [HBH96a], [HBH96b] an input application is distributed into parts for both the host and reconfigurable Xputer-based accelerators to achieve high performance. To reduce data transfer overhead and to exploit more parallelism, data scheduling has to take place, which is shown in the following. The paper is organized as follows:

Section 2 gives an overview about the target hardware and the reconfigurable Xputer architecture. Section 3 gives a brief overview on the Code-X framework. A new developed technique to schedule the data for improving the performance of the Code-X generated final task allocation and scheduling is presented in section 4. Finally, the paper is concluded.

2. The Xputer hardware

The new data-driven, non von Neumann paradigm of Xputers [AHR94] [HHS91] [Hirs91] is based on reconfigurable logic supporting structural programming. An Xputer provides an address generating device, to support structured data access for multi-dimensional arrays. This device is called data sequencer, in contrast to the instruction sequencer of a von Neumann processor. Note, that a sequence of instructions, like in the von Neumann paradigm, involves a tight coupling of the sequencer to the ALU, i.e. the instruction set is dependent on the ALU capabilities. So, if the ALU is changed, e.g. because the application demands additional capabilities, the instruction sequencer must change as well. In contrast to this, the data sequencer of the Xputer paradigm generates a stream of data, which is independent from the device which processes this data. Therefore, the ALU can be changed easily or, like in our prototype, made reconfigurable, so that the ALU capabilities can be adapted to the application.

An overview over the hardware structure of the MoM-3 is shown in figure 2. The accelerator consists of several Xputer modules (up to seven), connected by a special bus, the MoM-bus, which is connected to the host via an interface. Additionally to the modules, a controller (M3C, MoM-3 controller), which is used to configure the single Xputer modules. The modules run independently from each other and the host computer until a task is finished. The completion of a task is signaled by an interrupt, so the host can concurrently process other tasks in-between. A single Xputer module consists of three major parts:

- A local data memory
- A reconfigurable ALU (rALU)
- A data sequencer, implemented by a generic address generator (GAG)

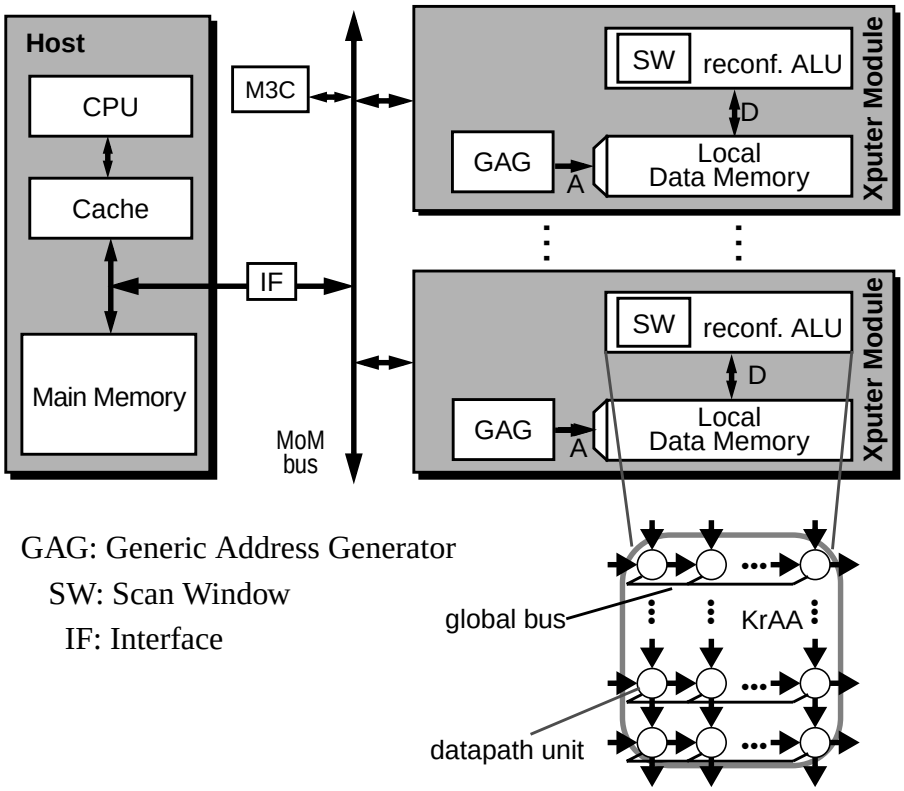

Figure 2. Hardware Structure of MoM-3 prototype

The local data memory is organized two-dimensionally to support applications working on two-dimensional arrays, but can also be interpreted to be higher dimensional by an appropriate mapping [Schm94]. Each memory can be accessed locally by the data sequencer of the same module, by the data sequencer of another module (via the MoM-bus), or by the host (via the interface and the MoM-bus). A local access by the data sequencer on the same module is fastest, as no bus-cycle is being used.

For the reconfigurable ALU (rALU), the Kress ALU Array (KrAA) [Kres96], formerly known as the reconfigurable Datapath Array (rDPA) is used. The KrAA comprises an array of reconfigurable processing elements called DPUs (data path units, see figure 2). The array can be made arbitrarily big. The DPUs are connected by a global bus as well as local connections between direct neighbours. The global bus is used to transfer data to the DPUs from outside the KrAA, i.e. the data memory. The local connections at the array borders can be connected together outside the array in different ways. So, torus, mesh or other structures are possible. Each DPU can be programmed to implement an operator, a routing function or both. The operators are 32 bits wide and include all operators of the programming language C. Therefore, the KrAA is well suited to be programmed in a high-level language. The

execution of the operators is transport-triggered, i.e. the operation starts as soon as all operands are available. So, pipeline structures can be realized easily. The KrAA can be reconfigured at run-time, also partially. The rALU contains also smart interface to the data memory called scan window (SW). The scan window is a kind of cache, which holds a rectangular part of the memory, which contains the data words needed by one step of the calculation. All data in the scan window can be accessed in parallel by the KrAA. Also, the scan window can be resized at run-time. An example configuration of the KrAA is shown in figure 3, together with an example scan window. In the example, the scan window is a 3 by

p_{02}	p_{12}	p_{22}
p_{01}	p_{11}	p_{21}
p_{00}	p_{10}	p_{20}

$$p_{11new} = p_{22}k_{22} + p_{12}k_{12} + p_{02}k_{02} + p_{21}k_{21} + p_{11}k_{11} + p_{01}k_{01} + p_{20}k_{20} + p_{10}k_{10} + p_{00}k_{00}$$

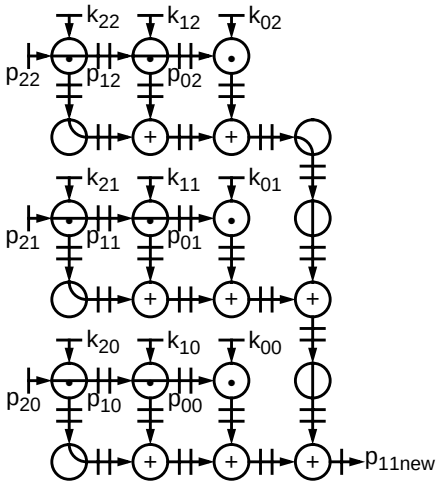

Figure 3. Example KrAA configuration and scan window

3 square. The KrAA configuration calculates a weighted sum of the nine values in the scan window and writes back the center value. If this configuration is applied to an image, it could be used to implement a FIR filter for image processing with adequate weights k_{ii}. It can be seen in figure 3, that the configured structure consists of three pipelines, which join together to calculate the result. The execution time in clock cycles for each operator on one DPU is known. Therefore, it is easy to calculate the total execution time of a complex structur, like the one in figure 3. The Datapath Synthesis System generates statistical data including this value.

The task of the data sequencer is to feed a data stream into the rALU. This is done by the GAG (generic address generator) [HBH96b], which generates an address sequence for the data memory, so that the data words get to the rALU in the correct sequence. For many applications, the access sequence for the data words follows regular patterns. The GAG exploits this situation by providing a hardware capable of generating a wide variety of access sequences, which can be described by only a few parameters. We call such an access sequence a scan pattern. Internally, the calculation of the address sequence is done in a pipeline consisting of two stages. This is shown in figure 4. In the first stage, the handle position generator inside the GAG calculates a new position of the scan pattern in the two-dimensional memory. This handle position is not used directly to fetch a data word from the resulting location. Instead, the whole scan window is moved to the handle position, allowing

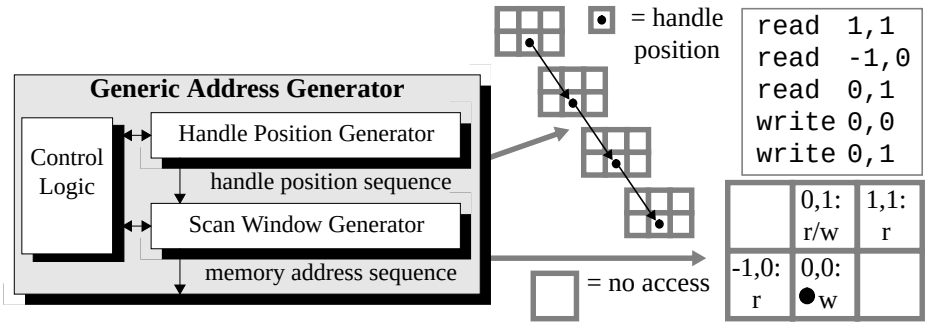

Figure 4. Internal structure of the MoM-3 GAG.

access to a whole area of data needed by one step of the calculation. In the second stage, the scan window generator generates the final address sequence for the data memory. Any access sequence to the data words inside the scan window can be programmed. For each data word, the access mode read, write, read / write or unused can be set.

So, an Xputer execution works as follows, considering the example in figure 4: First, the scan window is moved to the next position according to the scan pattern. In the example, the scan pattern is a simple sequence of linear movements, each 3 steps down and 2 steps to the right. Then, the data words at positions (1,1), (-1,0) and (0,1), relative to the scan window handle, are read and passed to the rALU in this sequence and the rALU starts the calculation. After the rALU is finished, two data words are written back to positions (0,0) and (0,1). Then, the scan window is moved again one step of the scan pattern. This process repeats until the scan pattern is completed.

Note, that the scan pattern describes explicitly the sequence of accesses to the data words, which can be realized by the GAG using few parameters. As the parameters include the step widths and boundaries of a scan pattern, it is possible to calculate the number of memory accesses of the pattern. With this information and the execution time of the complex operator in the KrAA, the total execution time of an Xputer task can be estimated.

For more details on the current Xputer hardware prototype MoM-3 and its applications please see [BHK95].

3. The dual Co-Design framework CoDe-X

For current and future Xputer prototypes a partitioning and parallelizing compilation framework CoDe-X is being implemented [HaBe97a], [HBH96a], [HBH96b]. CoDe-X is driven by the hardware parameters of the target Xputer prototype, and is based on two-level hardware/software co-design strategies and accepting X-C source programs (figure 5). X-C (Xputer-C) is a C dialect. CoDe-X consists of a profiling-driven 1st level partitioner, a GNU C compiler, and an X-C subset compiler. The X-C source input is partitioned in a first level into a part for execution on the host (host tasks, also permitting dynamic structures and operating system calls), and a part for execution on the Xputer prototype (Xputer tasks). Parts for Xputer execution are expressed in an ANSI C subset,

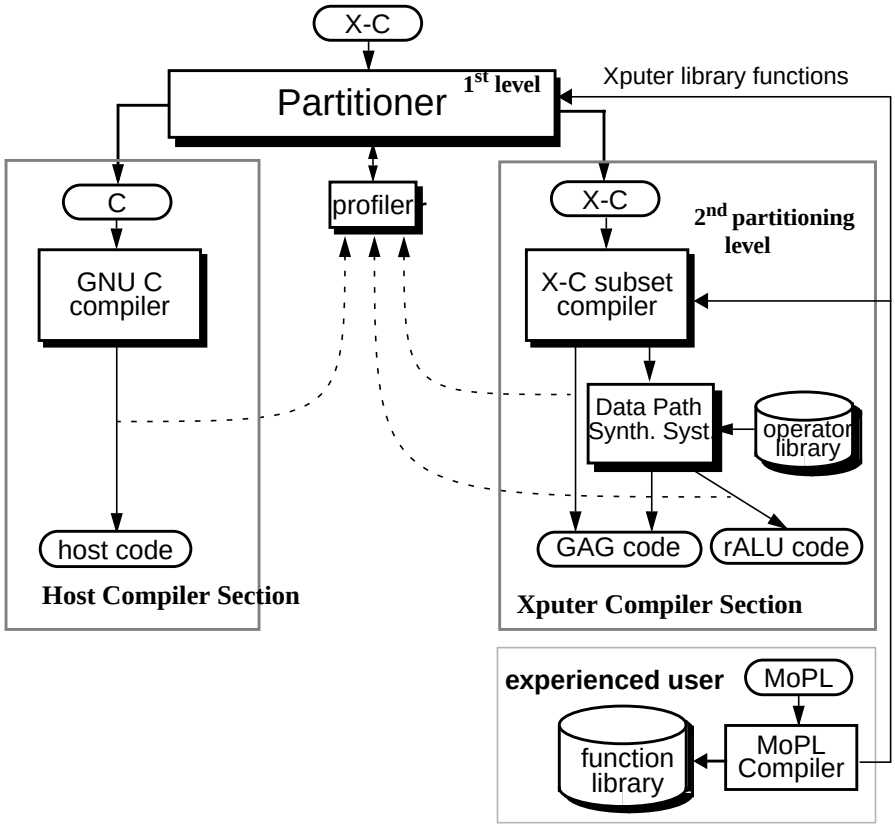

Figure 5. Overview on the Code-X framework

which lacks only dynamic structures [Schm94]. At second level this input is partitioned by the resource-driven X-C subset compiler in a sequential part for the data sequencer, i.e. the GAG, and a structural part for the rALU.

As X-C is an extension of C, experienced users may hand-hone their source code by including directly data-procedural MoPL code (Map-oriented Programming Language [ABHK94]) into the C description of an application. Also less experienced users may use special MoPL library functions similar to C function calls to take full advantage of the high acceleration factors possible by the Xputer paradigm (see figure 5). For more details about this optional data-procedural language extension and its usage see [HaBe97a].

Generally four kinds of tasks can be determined:

- host tasks which contain dynamic structures
- Xputer tasks,
- directly included MoPL code segments, and
- MoPL library function calls.

The host tasks have to be evaluated on the host, since they cannot be performed on the Xputer-based accelerators (because e.g. they contain host operating system calls). The Xputer tasks are the candidates for the performance analysis on the host and on the Xputer [HaBe97b]. The included MoPL-code segments as well as the MoPL library functions are used to get the highest performance out of the Xputer hardware. An experienced user has developed a function library with all library functions together with their performance values. These functions can be called directly in the input C-programs. They are evaluated in any case on the Xputer.

An Xputer job is represented by a task graph, on which a data dependency analysis based on the GCD-test (Greatest Common Divisor) [WoTs92] is carried out. Thus, tasks which can be evaluated in parallel can be found.

The X-C subset compiler [Schm94] realizes the resource-driven second level of partitioning and translates an X-C subset program (statical subset of ANSI C) into code which can be executed on the Xputer. The compiler performs a data and control flow analysis. First, the control flow of the program graph is partitioned according to the algorithm of Tarjan [Tarj74] resulting in a partial execution order. This algorithm is partitioning the program graph into subgraphs, which are called *Strongly Connected Components*. These components correspond to connected statement sequences like fully nested loops for example, which are possibly parallelizable. The Xputer hardware resources are exploited in an optimized way by analyzing, if the structure of statement sequences can be mapped well to the Xputer hardware avoiding reconfigurations or idle hardware resources [Schm94]. Xputers provide best parallelism at statement or expression level. So we try to vectorize the statement blocks in nested for-loops according to the vectorization algorithm of J. R. Allen and K. Kennedy [Kenn80], after a data dependency analysis has been performed [WoTs92].

4. Data scheduling in the Code-X framework

After the principles of the Code-X frameworkhave been illustrated briefly, we will take a closer look at its data distribution, called data scheduling. We will introduce a new extension to Code-X, which distributes the tasks and data of the application onto single Xputer modules and the host. As the host is handled the same way as the Xputer modules, we will not distinguish the host from Xputer modules, except for the time to access data memory. Our strategy avoids unnecessary copy operations between different data memories. The Xputer architecture supports our approach, as the execution time of a Xputer task can be estimated very well (see section 2).

As stated above, Code-X creates a task graph with a complete schedule for both host tasks and Xputer tasks, which shows the data dependencies between the single tasks. From this, parallely executable tasks can be derived.

As each Xputer module (and the host, of course) has its own memory, application data transfers occur between different memories. Additionally, there are different kinds of memory accesses, like shown in section 2, with different memory access times. Accesses to the local memory of the same module are faster than inter-module memory accesses.

To increase application performances, the memory transfers, i.e. copy operations between different modules should be minimized. For our approach, a performance analysis of the tasks is needed. We exploit the fact, that it is possible to calculate the number of memory accesses of a scan pattern as well as the total execution time of a task, like explained in section 2.

For illustration purposes, we assume a simplified schedule with discrete timesteps, like shown in the left part of the example in figure 6. We assume further, that the schedule

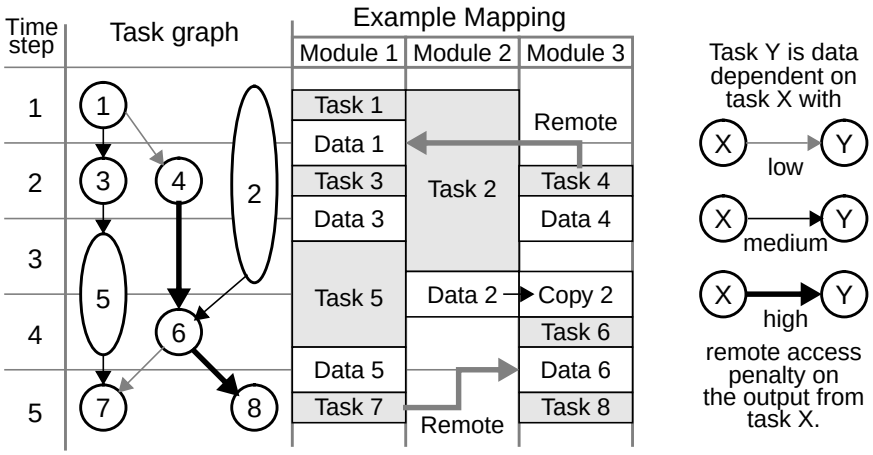

Figure 6. Data scheduling example for a simplified task graph

contains only Xputer tasks, which can be executed on the host or the Xputer, but are available as Xputer code.

We introduce the following values, which are used by our strategy: For a given task *T* and a data block *D*, let *scansteps(T,D)* denote the number of steps of the scan pattern and *windowsteps(T,D)* the number of read and write operations inside the scan window of this task on the data block. Both values can be calculated and are generated as statistical data by the X-C compiler and the Datapath Synthesis System, which is part of the Xputer compiler section (cf. figure 5). Then we can express the total number of memory accesses *accesses(T,D)* as:

Now, let t_{mem} be the time to access a data word in memory. Let $t_{MoM\text{-}bus}$ and $t_{hostbus}$ be the additional delay of the MoM-bus and the host bus respectively. We can write t_{local}, the time for a local memory access, t_{board} the time for an access to a different module and t_{host}, the time for an access from or to the host memory as:

$$t_{local} = t_{mem}$$

$$t_{board} = t_{mem} + t_{MoM\text{-}bus}$$

$$t_{host} = t_{mem} + t_{MoM\text{-}bus} + t_{hostbus}$$

The values t_{mem}, $t_{MoM\text{-}bus}$ and $t_{hostbus}$ are hardware-dependent. We will use the unified term t_{remote} for the time to access non-local memory. Depending, if the remote module is the host or an Xputer module, t_{remote} is either t_{board} of t_{host}.

Together with the size of a data block D, which we denote by *size(D)*, we can express the time to copy D from one module to another as:

Let *penalty(T,D)* be the additional time needed by a task T, if it has to access data block D remotely. Obviously, *penalty(T,D)* can be calculated by:

$$penalty(T, D) = (t_{remote} - t_{local}) \cdot accesses(T, D)$$

For a set of datablocks $DS = \{D_1,...,D_n\}$, the penalties sum up to the accumulated penalty *acpenalty(T,DS)*:

$$acpenalty(T, DS) = \sum_{Di \in DS} penalty(T, D_i)$$

The table in figure 6 shows a possible mapping for the example task graph given. For illustration purposes, we do not use numerical values for the penalty but only three degrees, „low“, „medium“ and „high“. Only if the penalty is „low“, a remote data access is bearable.

Using the above definitions, we will now describe our mapping-strategy.

The task graph from Code-X is processed timestep by timestep from the start to the end. In the first step, all tasks, which can be executed in parallel, are assigned to different Xputer modules. As Code-X performs a resource constraint scheduling, it is provided, that at any time, there are not more tasks scheduled than modules are available.

For all other timesteps, only the tasks, which start in that step are considered one by one. Let T_0 be the current task. There are several possible cases:

a) There is no predecessor to T_0. In this case, the task is assigned to a free module.

b) Task T_0 has one predecessor T_p, which has itself no other successors besides T_0. Here, T_0 is mapped onto the same module as T_p. Therefore, the application data can stay in the same module and needs not to be copied. In the example in figure 6, this applies to tasks 3, 5 and 4.

c) There are multiple predecessors to T_0, which have no additional successors. So, T_0 obviously needs data from several sources. Such a situation applies to task 6 in figure 6. In this case the *penalty(T_0,D_i)* is checked for every data block D_i needed by T_0, to find out, which data is most important. The task is then preferably mapped onto the module, which holds this data. Data blocks D_j from other modules are copied to the module of T_0, if *penalty(T_0,D_j)* is greater than *copytime(D_j)*. Otherwise, the data is accessed remotely.

d) Task T_0 has one or more predecessors $T_{p1},...,T_{pm}$, which have themselves successors $T_1,...,T_n$ besides T_0. Let *DS* be the set of all data blocks, which are needed by the $T_i \in \{T_0,T_1,...,T_n\}$. For each T_i and each $T_{pj} \in \{T_{p1},...,T_{pm}\}$ the accumulated penalty *acpenalty*(T_i,DS_{pj}) is calculated, where $DS_{pj} \subseteq DS$ is the set of data blocks T_i would have to access remotely, if it was not mapped onto the module of T_{pj}. All penalties are sorted and the task T_i with the highest value is mapped onto its prefered module. Then, the data blocks on this module are removed from the set *DS*. The penalties are calculated again and the process repeats, until all tasks are mapped.
After the mapping of the tasks, we can expect, that some data blocks are missing for each task. Therefore, for each Task $T_i \in \{T_0,T_1,...,T_n\}$ the set DM_i is determined, consisting of the data blocks needed by T_i, which are not available locally. Like in case c), a data block $D \in DM_i$ is copied to the local memory, if *penalty*(T_i,*D*) is greater then *copytime*(*D*).

5. Conclusions

A new data and task scheduling technique for improving the performance of reconfigurable Xputer-based accelerators has been presented. One or more Xputer processing modules with local memory can be handled. Multiple tasks and their data are mapped onto the processing resources, providing a high degree of parallelism. The mapping avoids unnecessary data transfers between modules.

Therefore, overall system performance can be increased by using the introduced data scheduling technique. The introduced MoM-3 hardware prototype allows the calculation of exact performance data, which is used to control the hardware/software partitioning process of the Code-X framework and the new scheduling method. CoDe-X allows the simultaneous programming of both the accelerator and the host by a two-level partitioning approach and a parallelizing compiler. The proposed scheduling technique is an extension to Code-X to optimize the task distribution incl. their data transfers, and to enhance the overall system performance.

6. References

[ABHK94] A. Ast, J. Becker, R. Hartenstein, R. Kress, H. Reinig, K. Schmidt: Data-procedural Languages for FPL-based Machines; 4th Int. Workshop on Field Programmable Logic and Appl., FPL'94, Prague, Sept. 7-10, 1994, Springer, 1994

[AHR94] A. Ast, R. Hartenstein, H. Reinig, K. Schmidt, M. Weber: A General Purpose Xputer Architecture derived from DSP and Image Processing. In Bayoumi, M.A. (Ed.): VLSI Design Methodologies for Digital Signal Processing Architectures, Kluwer Academic Publishers 1994.

[Ath92] P. Athanas: A Adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration; Ph.D. Thesis, Brown University, May 1992

[AtSi93] P. Athanas, H Silverman: Processor Reconfiguration Through Instruction-Set Metamorphis; IEEE Computer, Vol. 26, pp. 11 - 18; March 1993

[BRV93] P. Bertin, D. Roncin, J. Vuillemin: Programmable Active Memories: a Performance Assessment; DIGITAL PRL Research Report 24, DIGITAL Paris Research Laboratory, March 1993

[BHK95] J. Becker, R. W. Hartenstein, R. Kress, H. Reinig: High-Performance Computing Using a Reconfigurable Accelerator; Proc. of Workshop on High Performance Computing, Montreal, Canada, July 1995

[BeTe94] P. Bertin, Hervé Touati: PAM Programming Environments: Practice and Experience; IEEE Workshop on FPGAs for Custom Computing Machines, FCCM'94, IEEE Computer Society Press, Napa, CA, April 1994

[CCA95] H.A. Chow, S. M. Casselman, H. M. Alunuweiri: Implementation of a Parallel VLSI Linear Convolution Architecture Using the EVC1; IEEE Workshop on FPGAs for Custom Computing Machines, FCCM'95, IEEE Computer Society Press, Napa, CA, April 1995

[EVC94] N.N.: EVC-1 Info 1.1; Virtual Computer Corporation, 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

[GHK91] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, D. Lopresti: Building and Using a Highly Parallel Programmable Logic Array; IEEE Computer, Vol. 24, No. 1, IEEE Computer Society Press, January 1991

[HaBe97a] Reiner W. Hartenstein, Jürgen Becker: A Two-level Co-Design Framework for data-driven Xputer-based Accelerators; in Proc. of 30th Annual Hawaii Int. Conf. on System Science (HICSS-30), January 7-10, Wailea, Maui, Hawaii, USA, 1997

[HaBe97b] Reiner W. Hartenstein, Jürgen Becker: Performance Analysis in CoDe-X Partitioning for Structural Programmable Accelerators; to be published in Proc. of 5th Int'l. Workshop on Hardware/Software Co-Design CODES/CASHE'97, Braunschweig, Germany, March 24-26,1997

[HBH96a] Reiner W. Hartenstein, Jürgen Becker, Michael Herz, Rainer Kress, Ulrich Nageldinger: A Parallelizing Programming Environment for Embedded Xputer-based Accelerators; High Performance Computing Symposium '96, Ottawa, Canada, June 1996

[HBH96b] Reiner W. Hartenstein, Jürgen Becker, Michael Herz, Rainer Kress, Ulrich Nageldinger: A Partitioning Programming Environment for a Novel Parallel Architecture; 10th International Parallel Processing Symposium (IPPS), Honolulu, Hawaii, April 1996

[HBK95] R. W. Hartenstein, J. Becker, R. Kress, H. Reinig, K. Schmidt: A Reconfigurable Machine for Applications in Image and Video Compression; Conf. on Compression Techniques & Standards for Image & Video Compression, Amsterdam, Netherlands, March 1995

[HBK96] Reiner W. Hartenstein, Jürgen Becker, Rainer Kress: Custom Computing Machines vs. Hardware/Software Co-Design: from a globalized point of view; 6th International Workshop On Field Programmable Logic And Applications, FPL'96, Darmstadt, Germany, September 23-25, 1996, Lecture Notes in Computer Science, Springer Press, 1996

[HHS91] R. Hartenstein, A. Hirschbiel, K. Schmidt, M. Weber: A novel Paradigm of Parallel Computation and its Use to implement Simple High-Performance Hardware; Future Generation Computing Systems 7 (1991/92), p. 181 - 198

[Hirs91] A. Hirschbiel: A Novel Processor Architecture Based on Auto Data Sequencing and Low Level Parallelism; Ph.D. Thesis, University of Kaiserslautern, 1991

[Kenn80] K. Kennedy: Automatic Translation of Fortran Programs to Vector Form; Rice Technical Report 476-029-4, Rice University, Houston, Oct. 1980

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

[RLR93] F. Raimbault, D. Lavenier, S. Rubini, B. Pottier: Fine grain parallelism on a MIMD machine using FPGAs; FCCM'93, IEEE Worksh. on FPGAs for Custom Computing Machines, Napa, CA, April 1993; IEEE CS Press 1993

[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; FCCM'93, IEEE Worksh. on FPGAs for Custom Computing Machines, IEEE CS Press 1993

[Tarj74] R. E. Tarjan: Testing Flow Graph Reducibility; Journal of Computer and Systems Sciences 9 (1974), pp. 355-365

[WoTs92] M. Wolfe, C.-W. Tseng: The Power Test for Data Dependence; IEEE Transactions on Parallel and Distributed Systems, Vol. 3, No. 5, Sept. 1992