First International Workshop on
HyperTransport™ Research and Applications

Proceedings

of the

1st International Workshop on HyperTransport Research and Applications

WHTRA 2009

Editors
Holger Fröning
Mondrian Nüssle
Pedro Javier García García

ISBN: 978-3-00-027249-3

February 12th, 2009, Mannheim, Germany

University of Heidelberg, Computer Architecture Group

Proceedings of the
First International Workshop on HyperTransport Research and Applications (WHTRA2009)
Feb. 12th, 2009, Mannheim, Germany

Run-Time Reconfiguration for HyperTransport coupled FPGAs using ACCFS

Jochen Strunk*, Andreas Heinig*, Toni Volkmer*, Wolfgang Rehm* and Heiko Schick†
*Chemnitz University of Technology
Computer Architecture Group
Email: {sjoc,heandr,tovo,rehm}@cs.tu-chemnitz.de
†IBM Deutschland Research & Development GmbH
Email: schickhj@de.ibm.com

Abstract

In this paper we present a solution where only one FPGA is needed in a host coupled system, in which the FPGA can be reconfigured by a user application during run-time without loosing the host link connection. A hardware infrastructure on the FPGA and the software framework ACCFS (ACCelerator File System) on the host system is provided to the user which allow easy handling of reconfiguration and communication between the host and the FPGA. Such a system can be used for offloading compute kernels on the FPGA in high performance computing or exchanging functionality in highly available systems during run-time without loosing the host link during reconfiguration.

The implementation was done for a HyperTransport coupled FPGA. The design of a HyperTransport cave was extended in such a way that it provides an infrastructure for run-time reconfigurable (RTR) modules.

1. Introduction

With the emergence of dynamically and partially reconfigurable (DPR) FPGAs, the possibility to reconfigure partially reconfigurable regions (PRR) with run-time reconfigurable modules has appeared. This feature enables FPGA customers to change the design of a certain region of the FPGA during run-time while maintaining the full functionality of the remaining part. This new degree of freedom also facilitates system designers to develop single FPGA chip solutions where additionally required hardware, e.g. a peripheral interconnect, is also located inside the FPGA.

For host coupled FPGA systems, solutions are conceivable where a static part of the FPGA covers the host interface core and the remainder of the device can be reconfigured during run-time with one or more user specific application modules.

Such a FPGA system would offer continuous host link connectivity during the time of partial reconfiguration and would not depend on exclusive hot plug solutions, where the board, the BIOS and the operating system must support hot plug functionality, which is currently not the case for standard motherboards with operating systems like Linux and Windows.

Two distinct options for connecting FPGA accelerators to a host system do exist, either via a peripheral bus (e.g. PCI Express) or processor bus. Well suited for direct processor bus coupled FPGA systems are the AMD CPUs because of the open standard and low latency HyperTransport (HT) protocol.

Sharing the resources of a single FPGA between users is also imaginable. In a multi user or multi process environment several modules could be run simultaneously on the same FPGA if resources are sufficient.

Partial reconfiguration offers the chance of reducing implementation time of FPGA designs (rapid prototyping) if supported by the FPGA synthesis tools. Already functional parts could be left on the FPGA and only functionality under test is exchanged. It should be noted that this requires a strict modular overall design.

For highly available and real-time processing systems with host connection, run-time reconfiguration enables to exchange or to add functionality during system operation.

In the field of high performance computing nodes with FPGAs used as accelerators, run-time reconfiguration can be utilized to change offload compute kernels and to share FPGA device capacity.

Using FPGAs for acceleration, due to the creation of specialized processing engines utilizing the highly parallel nature of FPGAs, can lead to a significant reduction of compute time. A speedup of more than 50 compared to a CPU was achieved by Woods et al. [1] accelerating a Quasi-Monte Carlo financial simulation.

Zhang et al. [2] gained a speedup of 25 for another Monte-Carlo simulation.

To run such compute kernels on a single chip FPGA solution making use of the run-time reconfigurability, three main components have to be provided to the user. The first one is the operational infrastructure for running run-time reconfigurable modules (RTRM) on a host interface on the same FPGA. The second part consists of a framework which allows the user to build its own RTRMs. Last but not least, an generic interface must be provided to a user which offers functions for reconfiguration and communication between the host and the RTRM located inside the FPGA.
The rest of the paper is organized as follows:

Section 2 is devoted to related work. In section 3 capabilities of run-time reconfigurable FPGAs and the principles of creating partial configuration bit stream files are shown.

Section 4 describes the run-time reconfiguration support for a FPGA directly connected to AMD's processor bus. The enhancement for a HyperTransport cave implemented as host interconnect is shown. The infrastructure needed on the FPGA for the support of run-time reconfigurable modules and their creation is presented.

The software framework provided to the user is based on ACCFS (Accelerator File System) which is explained in section 5.

As proof of concept we have implemented two distinct compute kernel offload functions as run-time reconfigurable modules in section 6. The first RTR module acts as an offload function which finds patterns in a bit stream (pattern matcher) and the second module a Mersenne Twister generates pseudo random numbers at high output frequency.

Section 7 concludes the results of this paper.

2. Related Work

Utilizing RTR capabilities of FPGAs and building CPU coupled systems have been proposed under various aspects. Some are dealing with internal communication structures while others concentrate more on system integration.

A tool-flow for homogeneous communication infrastructure for RTR capable FPGAs was presented by Hagemeyer et al. [3] built upon the Xilinx design flow. In contrast Koch et al. [4] designed a framework named ReCoBus-builder without applying Xilinx's partial reconfiguration flow. Only Virtex-II and Spartan-3 FPGA are supported by the builder so far. Switch architectures with routers between RTR modules have been examined also in [5] [6].

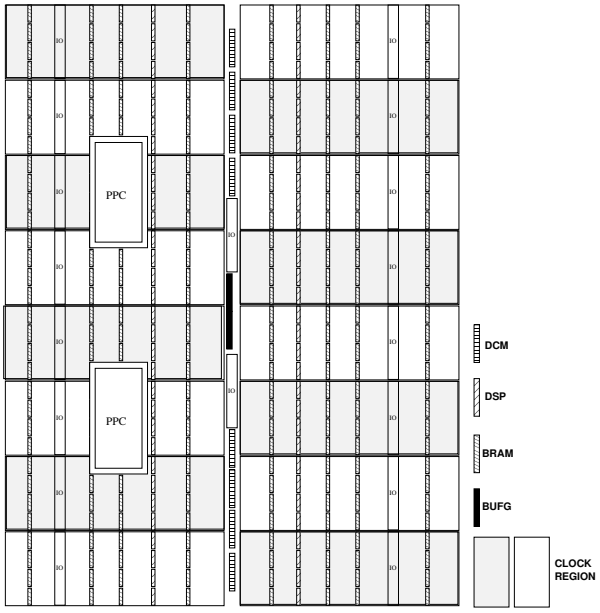


Figure 1. Schematic view of Xilinx XC4VFX60 FPGA

On the matter of integration of FPGA modules or threads for embedded systems different models have been proposed. ReconOS [7], a real time operating system implemented with static FPGA threads, is based on memory mapping and is used in embedded systems. Another model, BORPH [8], is based on the UNIX IPC mechanism and utilizes the integrated PowerPC as host.

For the integration of host coupled accelerators we proposed and implemented the Accelerator File System (ACCFS) [9]. This framework is based on the concept of a virtual file system. We have already shown the integration of the Cell/B.E. processor. In this paper we will show that ACCFS is best suited for the integration of FPGAs, even RTR capable FPGAs, into a host system.

3. Run-Time Reconfiguration on FPGAs

This section addresses the conditions which must be fulfilled, when using the feature of run-time reconfiguration on Xilinx FPGAs. These are important for the implementation of a HT cave which supports run-time reconfigurable modules.

3.1. Dynamic Partial Reconfiguration for FPGAs

This subsection is devoted to the dynamic partial reconfiguration (DPR) of Virtex-4 and Virtex-5

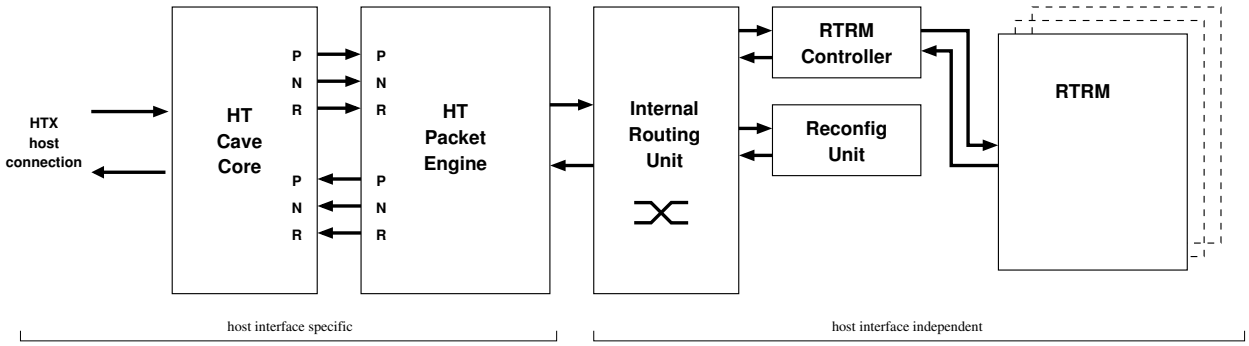


Figure 2. Infrastructure of HT cave with RTR support

FPGAs [10] from Xilinx which is one of the few manufacturers which offer DPR. The granularity of a partially reconfigurable region (PRR) is directly related to the configuration frames [11], which describe the function or contents of the slice containing LUTs or block RAM for example. The granularity in the height of a PRR matches the height of a clock region for Virtex-4 (16 CLBs) and Virtex-5 (20 CLBs). In the horizontal direction a PRR must begin with an even and end with a odd slice number. Figure 1 is a schematic view of the Virtex-4 XC4VFX60 FPGA used for the implementation of a HyperTransport cave supporting RTRMs described in the next sections. Note that we have a total of 16 clock regions available. For run-time reconfiguration three different interfaces are available, which are able to read the configuration bit stream of a RTRM. One of these is the JTAG port, which is a bidirectional serial host-clocked link. It is generally used for prototyping and debugging, working up to the speed of 24 MHz with available JTAG programmers. Another mode is SelectMAP, which works on a parallel interface connected to the physical IO pins of the FPGA achieving high throughputs. The third variant is the internal configuration access port (ICAP). It is an internal version of the external SelectMap working at a clock speed of up to 100 MHz at 32 bits width. For host coupled systems it is best suited, because it does not depend on external IOs and allows the shortest reconfiguration time.

3.2. RTR Modules and Design Flow

In this subsection the design flow is introduced for the creation of run-time reconfigurable modules. It also covers challenges and limitations of dealing with RTRMs. The design flow is based on ”Module based Partial Reconfiguration” [12] for Xilinx FPGAs. As a first step the HDL sources must be assigned either to the static part, which is constantly available during run-time, or the dynamical part. All communication between the two distinct parts has to go through hard macros, also known as bus macros. Clock resources and the hard macros must be instantiated in the HDL source and need to be assigned to a fixed location inside the FPGA. The run-time reconfigurable module itself is only instantiated as a black box, whose interface (entity) can not be changed during run-time. This means that a common interface must be created if other modules should be loaded in the partially reconfigurable region (PRR). The location and size of a PRR must be specified for the place and route process using the ”AREA_GROUP” constraint. It should be noted that for standard static design, neither PRR nor bus macros need to be specified. The same applies for the definition of the location of clock resources. To conduct the partial reconfiguration flow a patch is provided by Xilinx which must be applied to the standard synthesis tools.

4. Run-Time Reconfiguration Support for a HyperTransport Cave

For a single FPGA chip solution connected to a host utilizing HyperTransport as interconnect, it is essential not to loose the link during the time of the reconfiguration of a RTRM. This implies that the HyperTransport IP-core implementing a HT cave must be kept inside the FPGA as static part. Hot plugging is not supported so far by off the shelf systems. Even if the hardware is capable of handling such requests, most operating systems do not support this. Other RTRMs inside the FPGA would suffer also from the link loss. For that reason the HyperTransport cave is kept in the static region. In this section the enhancement of a HT cave is shown which provides an infrastructure for dealing with RTRMs.

```
entity rtrm is
port (
crq_c2m_addr     :in STD_LOGIC_VECTOR(31 downto 0);
crq_c2m_data     :in STD_LOGIC_VECTOR(31 downto 0);
crq_c2m_rq_valid:in STD_LOGIC;
crq_c2m_stop     :in STD_LOGIC;
crq_m2c_data     :out STD_LOGIC_VECTOR(31 downto 0);
crq_m2c_rp_valid:out STD_LOGIC;
crq_m2c_stop     :out STD_LOGIC;
crq_c2m_wr_rd    :in STD_LOGIC;

mrq_m2c_addr     :out STD_LOGIC_VECTOR(31 downto 0);
mrq_c2m_data     :in STD_LOGIC_VECTOR(31 downto 0);
mrq_c2m_rp_valid:in STD_LOGIC;
mrq_c2m_stop     :in STD_LOGIC;
mrq_m2c_data     :out STD_LOGIC_VECTOR(31 downto 0);
mrq_m2c_rq_valid:out STD_LOGIC;
mrq_m2c_stop     :out STD_LOGIC;
mrq_m2c_wr_rd    :in STD_LOGIC;

c2m_clk          :in STD_LOGIC;
c2m_res_n        :in STD_LOGIC;
m2c_intr         :out STD_LOGIC
)
end rtrm;
```

Figure 3. *Entity of RTRM*

4.1. RTR Infrastructure

A run-time reconfigurable infrastructure for a HyperTransport cave has to provide a communication mechanism between the host and the RTRM and perhaps between RTRMs themselves. It also has to comply to the rules of partial reconfiguration and the partial design flow. To ease porting the infrastructure to other interconnects, e.g. PCI Express, the functionality which must be implemented for a RTR infrastructure should be divided into two parts. One covers the host interconnect specific functions and the other the host interconnect independent portions.

The infrastructure designed for a HyperTransport cave supporting RTRMs consists of two host interface specific, i.e. HT Cave Core and HT Packet Engine, and four host independent parts, an Internal Routing Unit, a RTRM Controller, a Reconfig Unit and one or more RTRMs. The design of this infrastructure of a HT cave with RTR support is depicted in Figure 2. The HT cave design for the HyperTransport interconnect originates from [13]. The task of the HT Package Engine is to decode the HT packets coming from the host and to convert these into appropriate actions targeting the units inside the FPGA. This includes the creation of responses to requests from the host by injecting valid packets to the HT Cave Core. The Internal Routing Unit routes requests to and from internal units, e.g. RTRM Controller and Reconfig Unit. For fast run-time reconfiguration of RTRMs it is recommended to make use of an internal reconfiguration port. This is done by the Reconfig Unit which controls the internal configuration access port (ICAP) for Xilinx FPGAs. The Reconfig Unit itself is controlled by the vendor specific driver on the host, which validates if requests concerning the creation of new RTRMs can be served. The allocation of RTRMs to available RTR regions is also decided by the host system.

4.2. RTRM

Each RTRM has its virtual address space which is implemented 32 bits wide. This means that a global address space is not divided between the RTRMs using fixed addresses. It would be very difficult to resolve a request when two RTRMs demand the same fixed physical address for their memory regions which are exported to the user application using an entry in the virtual file system implemented on the host system.

The interface (entity) of a RTRM serves as an interconnect to the RTRM controller. Communication in both directions, i.e. controller requests (crq) and module requests (mrq), are possible using a stop and valid protocol. The entity of a RTRM in VHDL is shown in Figure 3.

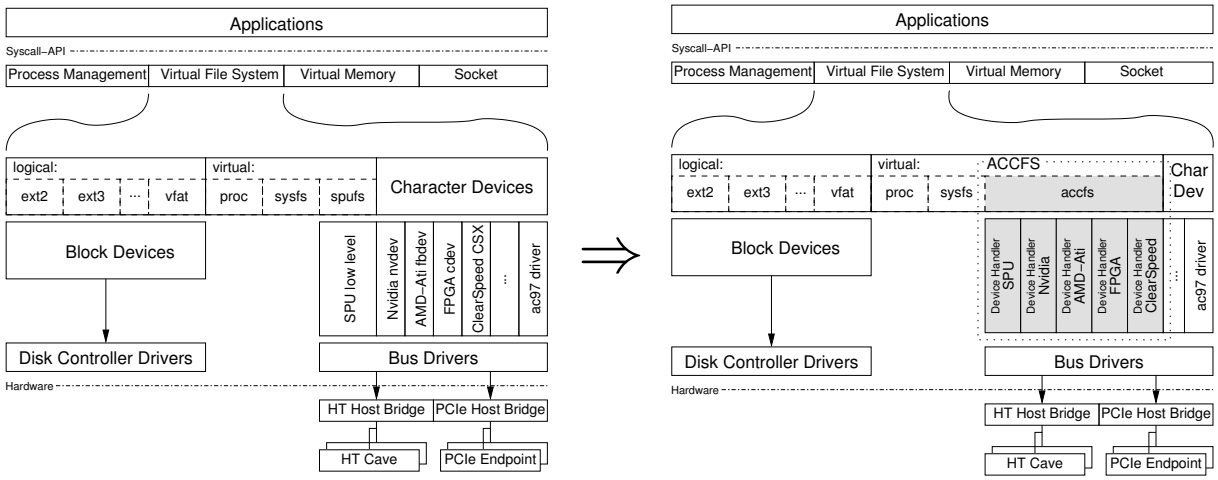
4.3. RTRM-Controller

The RTRM controller handles requests coming from the HT Core originated by the user application or from the RTRM itself. It converts physical addresses for directly accessing the RTRM, e.g. through direct load and store operations from the host to virtual RTRM addresses. The controller can also be used for RTRM to RTRM communication if desired.

4.4. Framework for a HT Cave supporting RTR

For generating the static part, i.e. the HT cave with RTR support, and the dynamical RTR modules, scripts are provided. The intention is to ease the creation of RTRMs for an application developer who is not so familiar with FPGA IP-core designs and run-time reconfiguration.

The top VHDL module is synthesized with the instantiated HT core, the HT packet engine, the internal routing unit, the RTRM controller and the Reconfig Unit by the `build_static` script. The RTRM module is only instantiated as a black box module. Then the static part is implemented with the partial flow option. While the user constraints file (ucf) normally contains location (LOC) constraints for external IO pins, this file must also contain additional LOC constraints for the PR flow covering all clock resources, in particular



(a) **Without ACCFS**

(b) **ACCFS: Common Generalized Interface**

Figure 4. ACCFS - Layered Structure

clock buffers and digital clock managers (DCMs). The resulting placed and routed design represents the basis for creating the dynamic configuration bit stream.

For the dynamic part, the user must supply an interface-compliant RTR module with the top entity name ”rtrm” and a description of the file entries which should be exported by ACCFS. This description consists of the type, the size and the virtual address which are essential to export the functionality to the user application. This additional information is added later to the final ACCFS configuration bit stream as a part of the header.

Using the `build_dynamic` script the user-supplied RTRM module is implemented with the partial flow option. Next, the Xilinx tools `PR_verify` and `PR_assemble` are used to build the partial bit stream file. Then the ACCFS RTRM bit stream file is created by adding header information containing the HT cave version, the FPGA board version and the user-supplied module description. Due to this header information, it is possible to transfer ACCFS RTRM bit stream files to other hosts which contain the same FPGA accelerator board and use the identical HT cave version.

5. ACCFS for Host System Integration

Different solutions exist for operating system integration of a FPGA. For example, BORPH [8] or ReconOS [7] provide a hardware process/thread abstraction which coexist beside ”normal” software processes. However, deep modifications of the Linux kernel are necessary to implement them. Furthermore, it is required to run Linux on the processing unit of the FPGA.

Due to the mentioned disadvantages we proposed and implemented the Accelerator File System (ACCFS) [9]. In this section we describe the major aspects of ACCFS for the integration of FPGAs into a host system. We start with a brief overview in subsection 5.1. Subsection 5.2 depicts the concepts of ACCFS. Thereafter, we present the integration steps for the HT-coupled Virtex-4 card in subsection 5.3.

5.1. Overview

ACCFS is an open generic system interface for the integration of different accelerator types into the Linux operating system. It is based on SPUFS (Synergistic Processing Unit File System) [14] which is used to access the Synergistic Processing Units of the Cell/B.E. processor. The goal of ACCFS is to replace the different character device based interfaces (cf. Figure 4a) with a generic file system based interface (cf. Figure 4b).

In the case of character devices the hardware functionalities are usually exported through the `ioctl` system call. However, this system call has the disadvantage of a non-standardized interface. Hence, the usage differs from one vendor to another.

In contrast, ACCFS defines a well structured `ioctl`-free interface based on a Virtual File System (VFS) approach. In Figure 4b the parts of ACCFS are shown as gray boxes. To be customizable when integrating new hardware ACCFS was split into two parts. Part one (”accfs”), provides the user interface,

and the other parts (”device handlers”) integrate the hardware.

Device vendors as well as library programmers benefit from ACCFS. Only the lowest abstraction levels have to be implemented inside the device handlers. The whole user interface is already provided by accfs. Thus integrating a new accelerator requires less device driver programming costs. The library programmer benefits from basic design concepts introduced in the next subsection.

5.2. Basic Concepts

In the previous subsection we already described the concept of **functionality separation** which eases the integration of new hardware. Another concept was the usage of a **VFS** which maps the accelerator to normal files. This enables us to implement a `ioctl` free and hence a nearly standard conform approach. All supported file I/O operations are POSIX conform with some exceptions. For example, it is not possible to write beyond the end of a file or to change the position of the current file pointer on some files.

ACCFS is designed to support the **virtualization** of the accelerators. We abstract the *physical accelerator* with an *accelerator context*. The context is the operational data set of the accelerator. It includes all information which are necessary to describe the current hardware state in such a way that the operation can be interrupted and resumed later without data loss. During the interruption another context is able to utilize the physical hardware. Virtualization optimizes the resource usage of the accelerators. Contexts which do not make use of the hardware at a given time are not scheduled on the physical accelerator.

Each context is bounded on a directory inside the VFS under the ACCFS mount point. The files inside this directory represent the functionalities of the accelerator. To support reconfigurable hardware the file set is **dynamically exported** and can change during runtime. For example, an additional memory can be exported due to reconfiguration of the FPGA with a new RTR module.

To interact with the accelerator several methods are feasible. One is the simple memory mapped IO with standard load/store machine instructions. In this direct memory access (DMA) method the host is the active part who issues a read/write for every memory access. Another method is DMA-bulk transfer. Here the accelerator needs a DMA unit capable of moving the data asynchronously to the host processor execution. In cases where the accelerator is able to initiate these transfers by itself, the DMA unit has to handle virtual memory managing issues, too. However, not every accelerator supports virtual memory. For this reason we restrict our solution to **host initiated DMA**, where the host setups the memory management unit and initializes the data transfer. The actual data movement is done asynchronously by the accelerator.

```
struct accfs_vendor
{
  int vendor_id;
  int (*create)(...);
  int (*destroy)(...);
  int (*run)(...);

          ...

  ssize_t (*memory_sdma)(...);
  ssize_t (*config_read)(...);
  ssize_t (*config_write)(...);
};
```

Figure 5. *struct accfs_vendor*

Finally, ACCFS supports **asynchronous context execution** based on an explicit synchronization primitive. This concept eases the software development because multi-threading is not required when using multiple accelerator units. Every context runs asynchronously to the host system. The finish status can be read through a ”status” file.

5.3. FPGA Support

To support HyperTransport coupled FPGA boards within ACCFS a new device handler has to be written. This device handler has to provide the structure *accfs_vendor* (cf. Figure 5). The first four entries has to be set and the others are optional. For example, if the callback function for the DMA-bulk transfer is not set (`memory_sdma`), accfs will use the internal routines to copy the data from/to the FPGA.

Further details of the device handler implementation are described in the reset of this subsection with the help of the typical FPGA usage model shown in Figure 6. An example code fragment using this model is shown in Figure 7 of section 6, where the case study is conducted.

5.3.1. Create Context. ACCFS enforces an accelerator based programming model. The main program is running on the host system and executes the compute kernel on the accelerator. To outsource such a kernel the application has to create a context by invoking the `acc_create` system call.

Currently our device handler does not support virtualization hence we can only exclusively provide the FPGA to one application.

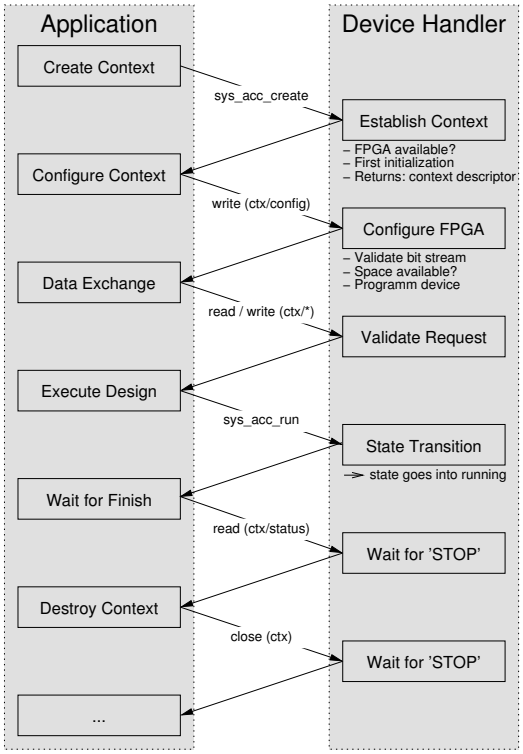


Figure 6. FPGA usage

5.3.2. Configuration. Loading the design is triggered by a `write` system call on the ”config” file. The data has to be a valid ACCFS bit stream. To ensure that the RTRM matches the RTR infrastructure we provide a tool chain which generates such a bit stream file by writing a special header before the bit stream data. The header contains all necessary information describing the bit stream such as the RTR capable core and FPGA board version. If the validation is successful, the FPGA is programmed with the configuration bit stream file using the internal reconfiguration port ICAP for Xilinx FPGAs or through an external JTAG programming device, e.g. Xilinx USB platform cable. After a successfully configuration the exported memories of the FPGA design are visible in the context directory.

5.3.3. Data Exchange. The access of FPGA memory is possible with the `read` and `write` system calls. In a later development stage these calls start a host initiated DMA-bulk transfer. If the memory is exported as memory mapped IO, the `mmap` system call will map the memory into the address space of the application.

The ”data exchange” operation is always possible after the configuration no matter whether the context is in execution or not.

5.3.4. Execute Design. To start the RTR module the application has to invoke the `acc_run` system call. The execution happens asynchronously, meaning that `acc_run` returns immediately. This enables the application to execute more than one context in parallel without using threads.

When the application needs to check the execution status, e.g. if the FPGA has finished its work, the ”status” file can be read. Unless this file was opened with *O_NONBLOCK* the `read` system call will block until the RTRM inside the FPGA has finished its task.

5.3.5. Destroy Context. When the application closes the file handle returned by `acc_create` the context gets destroyed.

6. Case Study of RTRMs for a HT Cave supporting RTR

6.1. Overview

As proof of concept we designed two different compute kernels as RTRMs for a HyperTransport coupled Xilinx Virtex-4 FPGA plug-in card [15]. The user program using the virtual file system ACCFS is able to configure and access the two RTRMs consecutively during the run-time of the user program at the time when they are needed. The first RTRM acts as an offload function which finds patterns in a byte stream (pattern matcher) and the second module, a Mersenne Twister, generates pseudo random numbers at high output frequency. For generating the appropriate partial bit stream files of the RTRMs the framework presented in subsection 4.4 is applied.

As hardware for the host system an Iwill DK8-HTX motherboard with two Opteron processors is utilized. The pre-installed BIOS is replaced by a customized LinuxBios version to get the HTX-card enumerated by the host system. The FPGA on the HTX card is a Xilinx Virtex-4 XC4VFX60.

6.2. RTRMs - Pattern Matcher and Mersenne Twister

Two RTRMs have been implemented, which are described in this subsection, a pattern matcher and a Mersenne twister based on the MT19937 algorithm [16].

The latter uses the MT32 [17] implementation, which is able to provide a new 32 bits pseudo random number each clock cycle. When the host performs a

```
int matcher_run (void * search_db_in, int db_size
    void * patterns_in, int pattern_count,
    void * results_out, int results_size) {
  int ret;
  char bufstatus[12];
  // create context of our static FPGA design
  int fd_ctx = (int)acc_create("example", V_ID,
      D_ID, 0750, NULL);

  // configure the design
  int fd_cfg = openat(fd_ctx, "config", O_WRONLY);
  configure_fpga(fd_cfg, MATCHER_RTRM_BITSTREAM);

  // open memory and status
  int fd_mem = openat(fd_ctx, "memory/FPGA MEM1",
      O_RDWR);
  int fd_status = openat(fd_ctx, "status",
      O_RDONLY);

  // fill memory with data (DMA bulk transfer)
  pwrite(fd_mem, search_db_in, db_size, DB_OFFSET);
  pwrite(fd_mem, patterns_in, 4 * pattern_count,
      PATTERN_OFFSET);

  // start the matcher
  acc_run(fd_ctx, 0);

  // check status
  // (wait until context execution finished)
  read(fd_status, bufstatus, 12);

  // read results of operation (DMA bulk transfer)
  ret = pread(fd_mem, results_out,
      results_size, RESULTS_OFFSET);

  // close files
  close(fd_mem); close(fd_status); close(fd_cfg);

  return ret;
}
```

Figure 7. Pattern matcher user program

read request on an arbitrary RTRM address, a new 32 bits number is provided.

The RTRM pattern matcher simultaneously compares several 32 bits patterns against a search database. The module consists of a finite state machine (FSM), four 32 bits comparators for each pattern, one control register, one status register as well as dual-port block RAMs for the search database, the search patterns and the results. Additionally, a 56 bits window is superimposed over the search database.

The registers and memories are mapped into the lower 27 bits addresses of the RTRM's address space and can be accessed by the host.

After the host has set the start bit in the control register, the FSM reads the search patterns from the pattern memory, the window is set to the beginning of the search database and the comparators are enabled.

Then, the first comparator of each search pattern tests the first 32 bits of the window, the second one 32 bits shifted by one byte, the third one 32 bits shifted by two bytes and the fourth the last 32 bits of the window against the search pattern. Hereby, the window can be shifted by 32 bits each clock cycle.

When the end of the search database has been reached, the results are written to the results memory. Afterwards, the 'finished' bit is set in the status register. Next, the host can read the matcher results from the results memory.

```
int run_compute_kernel (double * results_out,
    int results_count) {
  // create context of our FPGA design
  int fd_ctx = (int)acc_create("example", V_ID,
      D_ID, 0750, NULL);

  // configure the design
  int fd_cfg = openat(fd_ctx, "config", O_WRONLY);
  configure_fpga(fd_cfg, MERSENNE_RTRM_BITSTREAM);

  // open memory
  int fd_mem = openat(fd_ctx, "memory/FPGA MEM1",
      O_RDWR);

  // allocating buffer
  int32_t * buffer = (int32_t *) mmap(NULL,
      MEM_SIZE, PROT_READ | PROT_WRITE,
      MAP_SHARED, fd_mem, 0);

  int32_t * mt32_numbers = buffer + NUMBERS_OFFSET;

  // start the Mersenne twister MT32
  acc_run(fd_ctx, 0);

  // Example C function that uses random numbers
  c_kernel_function(results_out, results_count,
      mt32_numbers);

  // unmap buffer
  munmap((void *) buffer, MEM_SIZE);
  // close files
  close(fd_mem); close(fd_cfg);
  return 0;
}
```

Figure 8. Example that uses MT32 pseudo random numbers

6.3. User Application accessing RTRMs

The user function `matcher_run` (cf. Figure 7) demonstrates the usage of the RTRM pattern matcher. First, this function creates a new context and partially reconfigures the FPGA by the function `configure_fpga`. Then, the search database and search patterns are written to the RTRM's database and patterns memory using the `pwrite` system call. Next, the matcher is started using `acc_run` and the user function waits until the execution has finished. After that, the results are read from the FPGA into the buffer `results_out` by the `pread` system call.

The user function `run_compute_kernel` (cf. Figure 8) uses the pseudo random numbers generated by the RTRM Mersenne twister for the computation kernel `c_kernel_function`. This RTRM is initialized using the same functions like in the previous

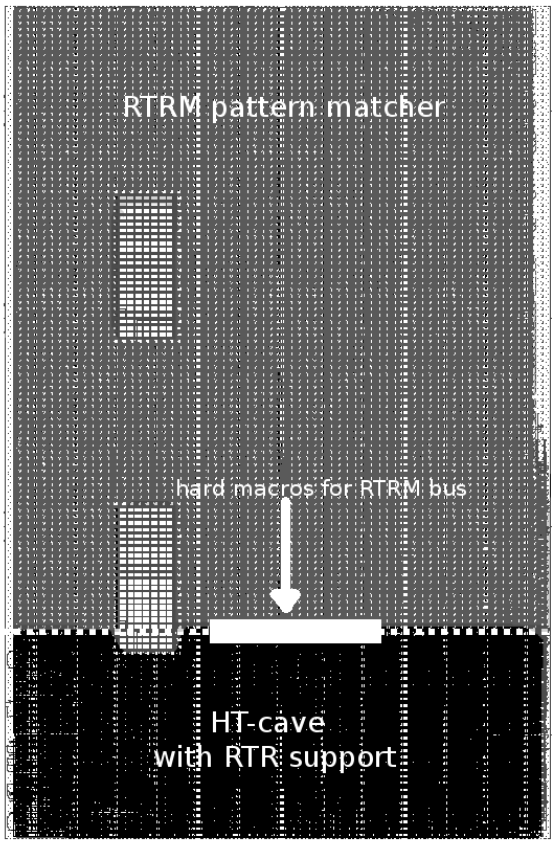


Figure 9. Placed and routed design of the HT cave with RTR support and the pattern matcher RTRM

example. In contrast to the previous one, the random numbers are not read using file handles, but can be accessed by the computation kernel via the memory-mapped buffer `mt32_numbers`.

6.4. Results of Case Study

The infrastructure for RTR modules based on the HT cave with RTR support was successfully implemented and verified. Furthermore, the virtual file system ACCFS was utilized for the integration and management of RTR modules on a HyperTransport plug-in card with a Xilinx Virtex-4 FPGA by using two example RTR modules which can be loaded onto the FPGA during run-time. For the implementation of the HT cave with RTR support at least 4 clock regions have to be reserved as static part.

The first RTR module acting as a offload function which finds patterns in a byte stream (pattern matcher) consists of 290 pattern matcher units resulting in a total of up to 116 billion 32 bits comparisons per second. This module nearly occupies all slices available within the clock regions designated for the RTRM. The placement is shown in Figure 9.

The second module implemented is a Mersenne Twister which generates pseudo random numbers at high output frequency.

For generating the partial bit stream file the framework presented in subsection 4.4 was applied.

7. Conclusion

By using the ability of run-time reconfiguration of FPGAs it is possible to build a single FPGA chip solution as a host coupled accelerator without loosing the host link connection during the reconfiguration of RTR modules. The design of a RTR infrastructure inside the FPGA was shown which allows to manage RTR modules during run-time. The implementation was done for FPGAs coupled directly to the HyperTransport processor bus of the host system. The concepts provided are applicable to other processor and peripheral bus coupled FPGAs. The software framework ACCFS, based on a virtual file system, provides a generic interface to user applications which is able to satisfy the demands of run-time reconfigurable computing.

8. Future Work

To speed up communication with high throughput between the host and a RTRM a memory transfer controller supporting bulk transfer between the different address spaces of the host and the RTRM should be implemented.

9. Acknowledgment

The project is performed in collaboration with the Center of Advanced Study Böblingen, IBM Research & Development GmbH, Germany.

References

[1] N. A. Woods and T. VanCourt, “FPGA Acceleration of Quasi-Monte Carlo in Finance,” in *Proceedings of the 2008 IEEE International Conference onField-Programmable Logic, FPL 2008, 8-10 September, Heidelberg*. IEEE, 2008, pp. 335–340.

[2] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, C. C. C. Cheung, D.-U. Lee, R. C. C. Cheung, and W. Luk, “Reconfigurable Acceleration for Monte Carlo Based Financial Simulation,” in *FPT*, G. J. Brebner, S. Chakraborty, and W.-F. Wong, Eds. IEEE, 2005, pp. 215–222.

[3] J. Hagemeyer, B. Kettelhoit, M. Koester, and M. Porrmann, in *Design of Homogeneous Communication Infrastructures for Partially Reconfigurable FPGAs (ERSA)*. CSREA Press, 2007.

[4] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs," in *Proceedings of the 2008 IEEE International Conference onField-Programmable Logic, FPL 2008, 8-10 September, Heidelberg*. IEEE, 2008.

[5] J. Surisi, C. Patterson, and P. Athanas, "An efficient run-time router for connecting modules in FPGAs," in *Proceedings of the 2008 IEEE International Conference onField-Programmable Logic, FPL 2008, 8-10 September, Heidelberg*. IEEE, 2008.

[6] T. Pionteck, C. Albrecht, K. Maehle, E., Hübner, M., and Becker, J., "Commuication Architectures for Dynamically Reconfigurable FPGA Designs," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, IPDPS USA, 2007.

[7] E. Lübbers and M. Planner, "ReconOS: An RTOS Supporting Hard-and Software Threads," in *Proceedings of the 2007 IEEE International Conference on Field-Programmable Logic and Applications*. Amsterdam: IEEE, 27-29 August 2007, pp. 441–446.

[8] H. K.-H. So and R. Bordersen, "File System Access From Reconfigurable FPGA Hardware Processes In BORPH," in *Proceedings of the 2008 IEEE International Conference onField-Programmable Logic, FPL 2008, 8-10 September, Heidelberg*. IEEE, 2008.

[9] A. Heinig, R. Oertel, J. Strunk, W. Rehm, and H. Schick, "Generalizing the SPUFS concept - a case study towards a common accelerator interface," in *Proceedings of the Many-core and Reconfigurable Supercomputing Conference*, Belfast, 1-3 April 2008.

[10] "Xilinx Virtex family," Website, 2008. [Online]. Available: http://www.xilinx.com/products/

[11] Xilinx, "Configuration Memory Frames," in *Virtex-4 FPGA Configuration User Guide (UG071)*, 2008.

[12] Xilinx, "Two Flows for Partial Reconfiguration: Module Based or Difference Based," in *Application Note: Virtex, Virtex-E, Virtex-II, Virtex-II Pro Families (XAPP290)*, 2004.

[13] D. Slogsnat, A. Giese, and U. Bruening, "A versatile, low latency HyperTransport core," in *Fifteenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2007.

[14] A. Bergmann, "The Cell Processor Programming Model," IBM Corporation, Tech. Rep., June 2005.

[15] M. Nuessle, H. Fröning, A. Giese, H. Litz, D. Slogsnat, and U. Brning, "A Hypertransport based low-latency reconfigurable testbed for message-passing developments," in *KiCC'07*, 2007.

[16] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.

[17] "Mersenne Twister, MT32. Pseudo Random Number Generator for Xilinx FPGA," Website, 2007. [Online]. Available: http://www.ht-lab.com/freecores/mt32/mersenne.html