General Purpose Computing with Reconfigurable Acceleration

Anthony Brandon, Ioannis Sourdis, and Georgi N. Gaydadjiev
Computer Engineering, TU Delft
Emails: {A.A.C.Brandon, I.Sourdis, G.N.Gaydadjiev}@tudelft.nl

***Abstract*—In this paper we describe a new generic approach for accelerating software functions using a reconfigurable device connected through a high-speed link to a general purpose system. As opposed to related ISA extension approaches, we insert system calls to the original program at hand to control the reconfigurable accelerator. The reconfigurable device is controlled by the host through a device driver, and initiates communication by raising interrupts; it further has direct accesses to the main memory (DMA) operating in the virtual address space. To do so, the reconfigurable device supports address translation, memory protection and paging, while the driver serves the device interrupts, and ensures that shared data in the host-cache remain coherent. The system is implemented in a machine which provides a HyperTransport bus connecting a Xilinx Virtex4-100 FPGA.**

I. Introduction

Integrating Reconfigurable computing with traditional Computing systems has been a challenge since the early days of FPGAs; several machines have been designed to this direction, such as PAM [1], Splash [2], and DISC [3]. The primary drawback of all these attempts was the limited bandwidth and increased communication latency between the host general purpose processor and the reconfigurable device. Recently, however, several components have been released which support standard high-speed communication between general purpose processor(s), memory and other peripheral devices. These solutions use high-speed on-board links such as the Intel QuickPath [4] and the AMD HyperTransport bus [5] and provide multi-GByte/sec low latency communication. The above developments offer a new opportunity for integrating reconfigurable computing in general purpose systems.

Rather than building a machine from scratch or suggesting fundamental architectural changes, it is more performance- and certainly cost-efficient to propose a generic solution that uses existing off-the-shelf components with only software (and configuration) modifications. As opposed to approaches that require ISA extensions to the host processor, such as Molen [6], Garp [7] and others, we describe a generic solution that requires only an FPGA device driver, a few compiler extensions and a reconfigurable wrapper in the FPGA.

The remainder of the paper is organized as follows: in Section II we describe the system architecture of our approach. In Section III, we discuss the details of our implementation platform and evaluate the performance and area cost of our solution. Finally, in Section IV we draw our conclusions.

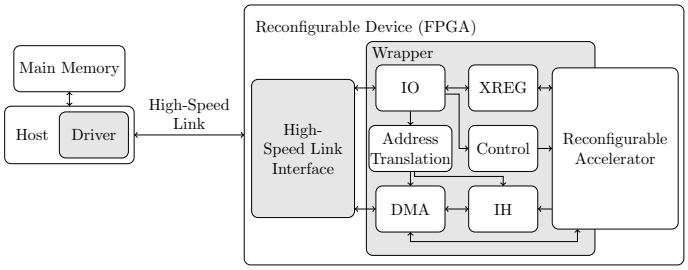


Fig. 1. Block diagram of the proposed system.

II. System Architecture

The proposed system consists of a general purpose machine with a reconfigurable device used for accelerating in hardware computationally intensive functions. Such functions are annotated in the code to indicate that they will be executed in hardware. The compiler is then responsible for generating the binary where the arguments of the respective functions are sent to the reconfigurable device and the return results are read from the device. In order to facilitate the execution of functions in hardware, the reconfigurable device works on **virtual shared memory**, supports **memory protection**, **cache coherence**, **paging**, and virtual to physical **address translation** maintaining a local TLB copy. Figure 1 illustrates the overview of the proposed system. The *reconfigurable device* is connected to the *host* general purpose processor and to the *main memory* through a *high-speed link*. On the host side, a *driver* has been developed to support the communication of the FPGA device with the host and the main memory. On the FPGA a module is used to *interface with the high-speed link*. The reconfigurable device further contains the *reconfigurable accelerator* intended to speed-up a software function; depending on the application running in the machine arbitrary accelerators can be designed. Finally, we have designed a *wrapper* around the reconfigurable accelerator to support the integration of the FPGA device in the system. More precisely, the wrapper controls the accelerator and performs tasks such as address translation, DMA operations, handles the interrupts caused by the device (IH), and maintains some memory-mapped IO regions: exchange registers (XREGs), the TLB copy, and control and status registers. The system supports three types of communication of the FPGA with the host and the memory. The host can write and read data from the FPGA, the FPGA can send data to the host, and the FPGA can read and write data to/from the memory.

A. Linux Driver

We have developed a driver that allows the software to use the reconfigurable FPGA device. The driver provides an *application programming interface* for software to interact with the device, it also performs the *initialization of the device* and *handles interrupts* caused by the device.

1) API: Application Programming Interface: The programming interface of the driver provides the following system calls: `open()`, `close()`, `read()`, `write()`, `ioctl()` used to control the functionality of the FPGA device. The `open()` system call is used by a program to get a lock on the device, while the `close()` system call releases the lock. The `write()` and `read()` system calls are used to write the arguments of function calls to the exchange registers (XREGs) of the FPGA device and to read the return values, respectively. Finally, the `ioctl()` call can be programmed to pass commands to the device, i.e., to initiate the execution of the reconfigurable accelerator. Using system calls rather than for instance extending the ISA of the host processor, like in [6] provides a more generic approach applicable to any system that supports a connection to an FPGA device.

2) Interrupts: The driver receives interrupts from the FPGA (i) on a TLB miss during address translation, and (ii) when the reconfigurable accelerator has completed execution. Since the High-Speed link interface supports only one interrupt identifier, we use a status device register to differentiate between the two cases above. The status register has different values for a TLB miss and for an execution completion interrupt.

a) Address Translation: During address translation at the reconfigurable device, a TLB miss may occur causing an interrupt. The device driver, then handles the interrupt providing the device with the missing TLB entry. During this process the driver should support *memory protection*. The reconfigurable device inherits the memory access permissions of the program that called it. The driver ensures that the FPGA device receives new TLB entries only for pages that has permissions to read or write; furthermore, read-only pages are protected from write accesses. Another task of the driver is related to *memory paging*. The driver checks whether a page accessed by the FPGA device exists in the memory; in case a page is swapped out to disk, the driver loads it back to the memory. Finally, the driver maintains *cache coherency* after the completion of a hardware-accelerated function.

b) Execution Completion: Upon execution completion of the reconfigurable acceleration the FPGA will raise an interrupt. Then, the driver will wake up the program that initiated the execution and will unlock all the pages that were locked and mapped during execution.

3) Initialization: The third task of the driver is to initialize the FPGA-device. When the driver is first loaded into the kernel, it registers the FPGA device-id and creates an interrupt-id. Then, in case the kernel detects the FPGA device, the driver is notified and initializes the device as well as several related data structures within the driver, e.g. it maps the memory mapped IO regions of the FPGA.

```
__attribute__\                  int foo(int a){
  ((user("replace")))             int b;
int foo(int a){                   write(dev, a, 0);
  int b;                          ioctl(dev, EXECUTE);
  ...                             b = read(dev, 1);
  ...                             return b;
  return b;                     }
}                               int main(void){
int main(void){                   dev = open(DEVICE);
  return foo(0);                  return foo(0);
}                               }
```

Fig. 2. Code modification by the extended compiler.

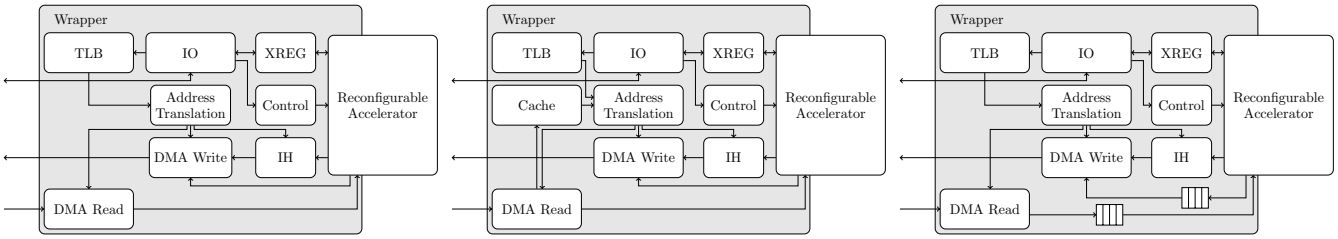
B. The Compiler

The programmer can indicate which functions are to be executed in hardware, by annotating them with gcc attributes: `__attribute__((user("replace")))`. The compiler will then automatically insert the appropriate system calls to the drivers API in order to achieve the hardware execution of the function. To do so, we have extended our gcc 4.5 compiler with a plugin which scans the source code for such annotated functions. The body of such a function is then replaced with the required system calls. As depicted in Figure 2, such system calls are the following: `open()`, `write()`, `ioctl()`, `read()` (`close()` is called by the kernel, hence not inserted by the compiler). As opposed to ISA extensions approaches, using system calls makes our approach generic. In addition, *supporting virtual addresses in the FPGA allows to pass as function-arguments pointers to larger data-structures in memory* which can then be accessed during the execution of the function with DMA transactions.

C. The FPGA Device

The FPGA Device consists of three parts: the high-speed link interface, the Reconfigurable accelerator and its wrapper. The high-speed link interface abstracts the communication details of the bus allowing the rest of the device to exchange data using a simpler protocol. The reconfigurable accelerator is the implementation of the hardware accelerated function and has a standard interface to exchange data with the rest of the system. The wrapper is located between the high-speed link interface and the accelerator; it is used to provide all the necessary functionality for integrating the FPGA device with the rest of the system; it supports address translation, DMA transfers and interrupts, maintains the memory-mapped IO device registers, and handless the interface with the driver.

a) Address Translation: The FPGA device is working on the virtual address space. Consequently, in order to access memory the device needs to have address translation support. To do so, the virtual addresses are translated into physical using a local TLB copy in the FPGA; our current implementation maintains a TLB copy of 512 entries. TLB misses are handled by interrupts raised by the device. In such case, the driver will write the missing entry to the FPGA TLB which then will be able to proceed with the respective translation. All pages stored in the FPGA TLB copy are then locked by the driver in memory, while pages the entries of which are replaced should get unlocked.



(a) Basic DMA support. (b) DMA manager with cache. (c) DMA manager with queues.

Fig. 3. Alternative designs for the accelerator wrapper, using different DMA managers.

b) DMA: Direct Memory Accesses: The FPGA device accesses the memory using DMA reads and writes. The Reconfigurable accelerator places the memory access requests and subsequently these are handled by the DMA manager module located in the wrapper. We have designed three different versions of the DMA manager, a basic one, a DMA with a cache, and a DMA with fifo queues. The basic DMA manager, shown in figure 3(a) serves the accelerator requests sequentially; this means that there are no concurrent DMA accesses and, in case of reads, the data read from memory need to arrive to the FPGA device before a new request takes place. This first approach introduces significant overhead since after each memory access the accelerator needs to stall waiting for the access to be completed. In our attempt to reduce the number of memory accesses we added a cache to the DMA manager which stores data coming to the device (Figure 3(b)). A DMA read will then bring an entire cache line to the FPGA and subsequent read requests to that line will not go to memory. In order to further reduce memory latency, we attempted to exploit the regularity of memory accesses in some types of applications; e.g. streaming applications. In such cases, the accelerator can pre-fetch data from the memory. The read data as well as the write requests should then be queued in a read and in a write fifo, respectively, as depicted in Figure 3(c). Multiple requests can be active at the same time while the reconfigurable accelerator continues processing. Furthermore, multiple memory accesses to consecutive lines can be merged into one request reducing the packetization overhead.

c) Interrupts: An interrupt caused by the FPGA is handled in the Interrupt Handler (IH) in the wrapper. Sending interrupts is achieved by writing to a memory location specified by the driver during the initialization of the device using a write DMA operation.

d) Memory mapped IO: The host processor controls the FPGA device through the driver using three regions of memory mapped IO. The host uses these regions by mapping them into the kernel address space and accessing them as arrays of data. The FPGA uses the IO regions as follows: The first region is mapped to the XREGs, which allow the driver to write function arguments to the device and read return values. The second region is used to send commands to the device and to get information about the status of the device. In this region every address is mapped to a different function e.g. for sending commands, reading the cause of an interrupt, and reading the address to be translated. The third region is used for the driver to read and write directly to/from the TLB.

TABLE I
IMPLEMENTATION RESOURCES (IN SLICES AND BRAMS) OF THE FPGA DESIGN BLOCKS. (S: SLICES, B: BRAMS)

	Basic DMA	DMA w/ Cache	DMA w/ Queues
HTX interface	5,066S + 29B		
Wrapper	865S + 7B	1081S + 11B	1,847S + 9B
Reconfigurable Accelerator	1,546S + 12B		
Total	7,477S + 48B	7,693S + 52B	8,459S + 50B

III. EXPERIMENTAL RESULTS

We use an AES program and its AES accelerator described in [8] to evaluate our approach. We experiment using the three design alternatives for our FPGA wrapper and compare them with each other as well as with running the purely software version of the program to the host processor.

The implementation platform used to instantiate the proposed system, consists of an AMD Opteron-244 1.8GHz 64-bit host processor and a 1-GByte DDR memory at an IWILL DK8-HTX motherboard. The motherboard provides a HyperTransport bus [5] and an HTX connector which we use to connect an HTX FPGA board [9]. The HTX FPGA board has a Xilinx Virtex4-100 FPGA. For the FPGA interface with the HyperTransport bus we used the HTX module developed in [10] which supports 3.2 GBytes/sec throughput.

Table I offers the area requirements of the FPGA modules. The HTX interface occupies about 5,000 slices and 29 BRAMs; that is due to multiple queues needed for the communication with the HyperTransport bus as well as tables to keep track of ongoing transactions. The wrapper with the basic DMA manager needs more than 800 slices and 7 BRAMs, when adding a cache to the DMA it needs about 1,100 Slices and 11 BRAMs, while the wrapper with the queues in the DMA needs roughly 1,800 Slices and 9 BRAMs. All wrappers need BRAMs for the TLB implementation while having queues or cache in the DMA adds 2-4 memory blocks. Although the rest of the design can operate at 200MHz, the operating frequency of the entire FPGA device is limited by the AES Reconfigurable accelerator to 100 MHz. It is noteworthy, that TLB misses take thousands of FPGA cycles, an address translation with a TLB hit requires 4 cycles, while the memory latency for a single read is 50 cycles.

We evaluate the performance of the three alternative designs of our system and compare with software using the ECB mode of the AES application having 256 bit keys and input files ranging from 16 Bytes to 64 MBytes [8]. The above configuration of the AES reconfigurable accelerator processes 128-bits every 16 cycles on a cycle time of 10 ns, having

a processing throughput of 80 Mbytes/sec. Although the bus supports IO throughput of 3.2 GBytes/sec, the above means that our accelerated function is compute bounded.

Figure 4 shows the AES execution time of the four alternatives for different file sizes to be encrypted, while Figure 5 shows the speedup of the hardware approaches compared to software. For small file sizes the software version is significantly better, up to 5× faster than the FPGA accelerated approaches. However, when the file size gets larger than 1KByte then accelerating the core AES function in hardware gets faster. For 64 MByte files the FPGA accelerated version is 1.5-5× better than software. As depicted in the results, the basic DMA configuration is always slower than in software; practically the memory latency is dominating and the reconfigurable accelerator is most of the time waiting for IO rather than processing. The DMA version with cache is up to 2× faster than the basic DMA and up to 1.7 better than software. In this case, several memory accesses are avoided; a memory access for a single word (64-bits) brings to the FPGA cache a cache line of 8 words (512-bits) which may potentially give subsequent cache hits. The DMA with queues is able to improve about 3× the DMA with cache for large file sizes and up to 5× compared to software. In the DMA with queues, the memory latency is almost totaly hidden since data are prefetched and stored in queues before they are consumed by the accelerator, while both read and write requests are packetized more efficiently allowing a single command to handle memory accesses for more than one words. Another interesting note is that for large files (64 MBytes) the design with Queues in the DMA manager supports a processing throughput of about 80Mbytes/sec which is equal to the theoretical maximum throughput. Finally, as expected, the larger the input sizes the better the performance of the system, since the setup overhead at the beginning of the program is then hidden by the long processing of the accelerated function.

In general, attempting to hide the memory latency is very crucial for system performance. Adding queues in the DMA can be applied in certain types of applications, such as streaming ones, due to their regular memory accesses. A cache next to the DMA seems to be the alternative for functions that have a more irregular memory access pattern. Finally, the overhead of integrating an FPGA card in a system as proposed in this paper is 10-15% in a Xilinx Virtex4-100.

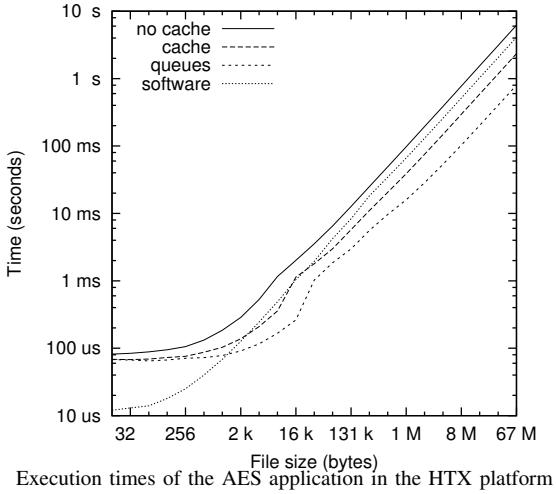


Fig. 4. Execution times of the AES application in the HTX platform when running in software, and with hardware acceleration.

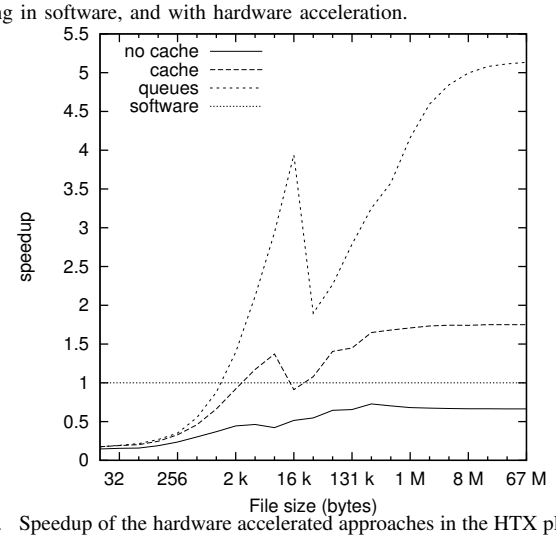


Fig. 5. Speedup of the hardware accelerated approaches in the HTX platform compared to purely software.

IV. Conclusions

We described a new generic, platform-independent approach for exploiting reconfigurable acceleration in a general purpose machine. In this paper we showed techniques to integrate a reconfigurable device into systems that provide a high-speed link to an FPGA device. We developed a Linux driver to control the device and support interrupts. Furthermore, we extended the gcc compiler to insert system calls to the code for the control of the reconfigurable accelerator. The proposed system offers direct access of the FPGA to the shared main memory of the machine using virtual addresses, memory protection, paging, and coherent cache of the host. We described three alternative designs for the DMA manager of the FPGA device. Finally, we evaluated the proposed system using an AES application and showed that significant speedup can be achieved over the pure software implementation.

References

[1] P. Bertin and H. Touati, "Pam programming environments: practice and experience," in *IEEE Workshop on FPGAs for Custom Computing Machines*, April 1994, pp. 133 –138.

[2] J. M. Arnold, "The splash 2 software environment," *J. Supercomput.*, vol. 9, no. 3, pp. 277–290, 1995.

[3] M. Wirthlin and B. Hutchings, "A dynamic instruction set computer," in *IEEE FPGAs for Custom Computing Machines,*, 1995, pp. 99 –107.

[4] "Intel quickpath," http://www.intel.com/technology/quickpath/.

[5] "Hyper transport bus," www.hypertransport.org.

[6] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Transactions on Computers*, vol. 53, pp. 1363–1375, 2004.

[7] J. Hauser and J. Wawrzynek, "Garp: a mips processor with a reconfigurable coprocessor," in *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997, pp. 12 –21.

[8] R. Chaves, G. Kuzmanov, S. Vassiliadis, and L. Sousa, "Reconfigurable memory based aes co-processor," in *13th Reconf. Arch. Workshop*, 2006.

[9] "Htx broard," http://ra.ziti.uni-heidelberg.de/index.php?page=projects&id=htx.

[10] D. Slogsnat, A. Giese, M. Nüssle, and U. Brüning, "An open-source hypertransport core," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 3, pp. 1–21, 2008.