Reconfigurable Hardware Module Sequencer - A Tradeoff Between Networked and Data Flow Architectures

Kai-Jung Shih[†], Chin-Chieh Hung, and Pao-Ann Hsiung[‡]
Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi, Taiwan-62102, ROC
[†]kjshih@cs.ccu.edu.tw, [‡]hpa@computer.org

Abstract

Dynamically reconfigurable systems either adopt a processor-controlled networked architecture or a sequencer-controlled data flow architecture. In the networked architecture, the processor is overloaded with data transfer requests, whereas in the data flow architecture, the burden is completely shifted from the processor to the data sequencer. As a tradeoff between these two extremes, this work proposes a novel module sequencer architecture, which not only allows the processor and the sequencer to share the heavy data communication load, but is also more coherent with the conventional processor-FPGA architecture. Further, the architecture is highly flexible because it can be tuned to fit a particular application. Application examples show how the proposed architecture is superior to the networked architecture in terms of lower communication load and to the data flow architecture in terms of reduced system complexity.

1. Introduction

With technology progress, the advent of the FPGA represents a trade off between performance and flexibility. Given the large amount of resources, *Dynamically Partially Reconfigurable Systems* (DPRS) can now be implemented in a single FPGA [5]. Unlike von-Neumann based architectures, there are currently no standard memory hierarchy and communication schemes for DPRS. However, two communication architectures are commonly adopted, namely *processor-controlled network architecture* (PNA) and *sequencer-controlled data flow architecture* (SDA). The main problem in PNA is that the processor is easily overloaded with too many communication requests. The challenge in SDA is that the high complexity in generating low-level data flow instructions makes optimization difficult and thus it is not easy to achieve high communication performance.

As a tradeoff between the low communication performance of network architectures and the high complexity of data flow architectures, a novel module sequencer architecture (MSA) is proposed in this work, which solves the issues related to reduced communication overhead, simplified programming model, simplified bus architecture, and virtual function mapping.

This article is organized as follows. Section 2 discusses related research work and compares them with our architecture. The proposed module sequencer architecture is described in Section 3. The illustration examples are given in Section 4. Finally, conclusions are described in Section 5.

2. Related Work

Instead of describing the generally well-known bus or NoC based PNA, we focus on two typical SDA in this section. *Transport Triggered Architecture* (TTA) [1][2], was proposed for customizing application-specific instruction-set processor (ASIP) designs. The TTA is a static hardware with simple design that moves the application complexity from hardware to software or the compiler design. *Reconfigurable Pipelined Datapaths* (RaPiD) [3][4] is a domain-specific coarse-grained reconfigurable architecture. RaPiD is a typical data flow architecture with a data sequencer.

3. Module Sequencer Architecture

Similar to other DPRS architectures, the target module sequencer architecture has a statically configured part and a dynamically reconfigurable part. As shown in Figure 1, the static part consists of a microprocessor, RAM, static hardware accelerators, a configuration device, and the proposed *Reconfigurable Module Sequencer* (RMS). Unlike other architectures, the dynamic part is controlled by the RMS.

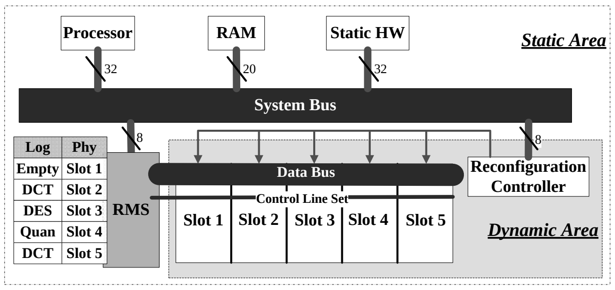

Figure 1. Module Sequencer System Architecture

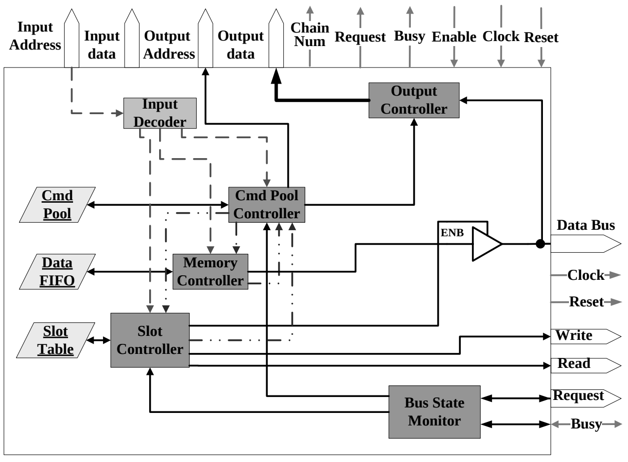

Figure 2. Reconfigurable Module Sequencer Architecture

Each kind of reconfigurable hardware function block is associated with a unique logical ID, which is mapped to a physical slot ID by the RMS dynamically.

The proposed MSA is an efficient blending of the conventional PNA and SDA architectures, because the microprocessor and the RMS share the data communication and control workload in a running application.

In MSA, an application is defined by a set of partially ordered command sequences called chains. Each chain $\langle f_0, f_1, \ldots, f_n, f_{n+1}\rangle$ consists of a sequence of $n+2$ functions, where f_0, f_{n+1} are software functions and $f_1, \ldots, f_n$ are hardware functions. A data transfer request is defined by a pair of functions $\langle f_{i-1}, f_i\rangle$.

3.1. RMS Design

As illustrated in Figure 2, the RMS has nine components, including three internal storages, four controllers, a bus state monitor, and an input decoder. The storages include a command pool (CP) that stores the chain commands, a data FIFO (DF) that caches the input data, and a slot table (ST) that records the state information for each slot. The state of a slot includes the mapping between logical and physical ID, the usage status, and the execution status if it is configured. The command pool controller (CPC) accesses the CP, stores chains into it, and selects an enabled data transfer request to be executed from some chain. The memory controller (MC) loads input data from DF to the configurable data bus. The slot controller (SC) accesses ST and controls the reconfigurable bus by asserting and deasserting the control signals. The output controller (OC) that sends output data to the processor through the system bus. The bus state monitor (BSM) checks the state of the reconfigurable bus, and dispatches signals to different controllers. The input decoder decodes command type and sends data or commands to the different controllers.

3.2. RMS Control Flow

The interaction between the RMS and the processor is triggered when the processor sends three types of commands to the RMS. The ID mappings are stored in ST by SC, the chain commands are stored in CP by CPC, and the input data are stored in DF by MC. The CPC checks for data transfer requests in CP and selects a request belonging to the chain with highest priority. To execute a data transfer request, the CPC queries the SC to check if the hardware of the requested function is configured in some slot and queries MC to check if its input data are available in the DF if it is the first data transfer request. If the responses from the SC and MC are both positive, then the CPC notifies the SC to assert the read and write control signals of the corresponding functions and the MC to transfer data. Otherwise, execution is postponed if the requested hardware function or the data of the selected chain are unavailable. In this case, the CPC selects another request to execute.

When the SC receives a function query signal from the CPC, it refers to the ST to check if the corresponding functions are configured. If configured and unused, SC acknowledges that the requested function is ready. When the SC receives a function execution signal, it asserts the write signal of the sender and the read signal of the receiver. If the sender is SW, the SC enables the tri-state buffer as shown in Figure 2, so that the data of the chain sent by the MC can be transferred on the bus. If not configured or all physical instances are busy, the SC acknowledges microprocessor that the requested function is unavailable.

When a data transfer is finished, the corresponding functions assert the busy signal indicating that the data bus is free for another transaction. The CPC initiates the execution of another data transfer request.

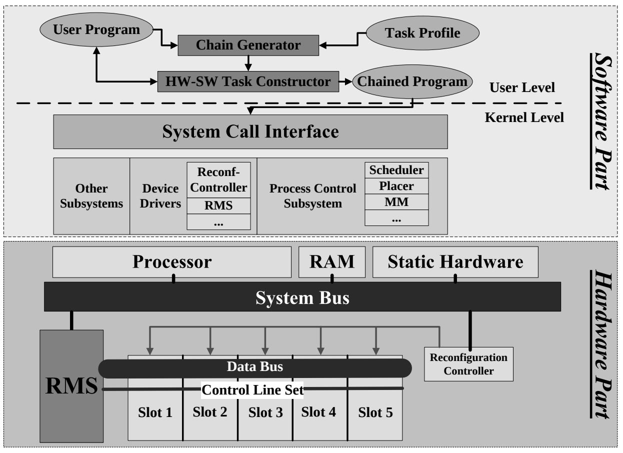

Figure 3. Programming Model and Chained Program Operating System

3.3. Programming Model

The programming model for MSA tries to follow a conventional one so that a user need not learn a new programming method. As shown in Figure 3, given a user program and corresponding task profile information, the chain generator determines the hardware-software partition. The hardware-software task constructor reorganizes the user program by replacing selected loops with RMS driver system call invocations and synchronization and buffering constructs. The result is a modified program called the *chained program*.

To support the execution of chained programs in MSA, an operating system for chained programs (CPOS) is required. Besides being an operating system for reconfigurable systems (OS4RS), CPOS has a system call that allows chained programs to send a request for executing a chain through the RMS driver. CPOS also has a hardware-software task scheduler, a hardware function block placer, a driver for the configuration controller, and other I/O device drivers. It must be noted here that the allocation, management, placement, and scheduling of reconfigurable hardware function blocks are all performed by CPOS, which means the RMS is only responsible for executing a chain request by coordinating the data transfers between blocks and between the processes and the blocks. The development of CPOS is still an on-going work and requires further design and implementation.

3.4. Performance Model

To evaluate the effectiveness of our proposed RMS architecture, we will compare RMS with the processor-controlled network architecture. Since we are improving the communication load for processors, one might wonder if existing schemes such as DMA would suffice. However, we will show that, as expected, DMA is effective only when the data size is very large and not if there are lots of communications. For evaluating the performance in executing a task consisting of k iterations of chain $\langle f_0, f_1, \cdots, f_n, f_{n+1}\rangle$, where f_i is the i^{th} function, f_0 and f_{n+1} are software, f_1 to f_n are hardware, and n is the total number of hardware functions, we first define the following notations.

- t_i: Data transfer time from f_i to f_{i+1} in cycles,
- DT_S: DMA setup time in cycles,
- DFS: RMS Data FIFO size in bytes, and
- SZ: Total input data size of the chain in bytes.

We assume the context switch time to be negligible in the following evaluation. We compare four different architectures depending on the use of RMS and DMA. The number of cycles a processor must expend in handling data communication for the task is as follows.

- No RMS, No DMA:

$$k \times \sum_{i=0}^{n} t_i \tag{1}$$

- No RMS, With DMA:

$$DT_S \times k \times (n+1) \tag{2}$$

- With RMS, No DMA:

$$(t_0 + t_n) \times k + (n+2) \tag{3}$$

- With both RMS and DMA:

$$DT_S \times \lceil SZ/DFS \rceil \times 2 + (n+2) \tag{4}$$

4. Experiments

The target module sequencer architecture was modeled, designed, and implemented. However, for performance evaluation we developed a SystemC-based simulation framework for the proposed architecture. The RMS cache size is 4096 byte, and the DMA setup time is 15 cycles. The first example was used for checking feasibility of the proposed architecture. It has three reconfigurable hardware functions and the chain command is $\langle f_0, f_1, f_2, f_3, f_4\rangle$, where f_0, f_4 are software functions. The total input data size is 2048 bytes. Each data transfer size between two successive hardware functions and between the processor and the first/last function is 64 bytes, which take 16 bus cycles to communicate. The total number of iterations for the chain is 32.

Table 1. Processor Cycles for Handling Communication

#	RMS	DMA	Toy	JPEG	DES
A	NO	NO	$2,048$	$510,000$	$1,024$
B	NO	YES	$1,920$ -6.25%	$450,000$ -1.18%	960 -6.25%
C	YES	NO	$1,029$ -49.76%	$150,005$ -70.59%	$1,027$ $+0.29\%$
D	YES	YES	35 -98.29%	$3,545$ -99.31%	35 -96.58%

Table 2. Execution Time of JPEG and DES With Different Priority

#	JPEG	DES	JPEG Time	DES Time	System Time
1	YES	NO	$1,598$	N/A	$1,598$
2	NO	YES	N/A	$2,062$	$2,062$
3	YES	YES	$1,536$	$1,814$	**3,660**
4	Low	High	$2,018$	$2,498$	**2,498**
5	High	Low	$1,853$	$2,753$	**2,753**

Low and High Are Priorities of JPEG and DES

The second example is an encrypted transmission of compressed images, which contains two chains, namely the JPEG [7] chain and the DES [6] chain. The JPEG chain is $\langle SW, DCT, Quantization, Entropyencoder, SW\rangle$ and the total input data size of the JPEG chain is $480,000$ bytes. The DES chain that takes 64-bit data as information and 64-bit data as key. The total data for this chain is 1024 bytes for data and 1024 bytes for key. A single execution takes 18 cycles to encrypt 64-bit data. The DES chain to be executed is $\langle SW, DES, SW\rangle$.

As given in Table 1, we compare the total number of cycles expended by the processor for handling communication in four different architectures for each application example. We can observe that the RMS brings very good performance improvements (70%) than the DMA (6%) when the chain is time-consuming such as the JPEG encoder. The RMS does not work well for small chains such as the DES because our architecture tries to reduce the communication between the reconfigurable hardware modules of a chain. The combination of DMA and RMS results in the best performance improvements ($96\% \sim 99\%$) in all cases because a single DMA setup can transfer only a limited amount of data, whereas with the help of the data FIFO buffer in RMS, a single DMA setup can transfer as much as the data FIFO can accommodate with whole iterations of a chain.

In Table 2, we compare the total system execution time for 5 configurations: (1) a single JPEG chain, (2) a single DES chain, (3) a JPEG with DES chained sequentially, (4) a low priority JPEG chain running in parallel with a high priority DES chain, and (5) a high priority JPEG chain running in parallel with a low priority DES chain. Data size for JPEG and DES are both $1,024$ bytes. Comparing configurations (3), (4), (5), we observe that configuration (3) has the worst performance because the functions are executed sequentially, in a single chain. Compared to configuration (5) and all other configurations, configuration (4) gives the best performance. This is because the most time consuming chain such as DES here is given the highest priority in RMS.

5. Conclusions

We proposed a novel module sequencer architecture as a tradeoff between networked and data flow architectures. Experiments show that the proposed architecture reduces the heavy communication load for processors by as much as 99% and also reduces the high programming complexity found in data flow architectures. Future work will consist of support for preemptive hardware functions and the collaboration of RMS with the scheduler and placer in an OS.

References

[1] H. Corporaal. Design of transport triggered architectures. In *Proceedings of the 4th Great Lakes Symposium on Design Automation of High Performance VLSI systems*, pages 130–135, March 1994.

[2] H. Corporaal and H. Mulder. Move: A framework for high-performance processor design. In *Proceedings of the IEEE conference on Supercomputing*, pages 692–701, 1991.

[3] D. Cronquist, P. Franklin, C. Fisher, F. M., and E. C. Architecture design of reconfigurable pipelined datapaths. In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pages 23–40, March 1999.

[4] C. Ebeling, D. Cronquist, P. Franklin, J. Secosky, and S. Berg. Mapping applications to the rapid configurable architecture. In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, pages 106–115, April 1997.

[5] C.-H. Huang, K.-J. Shih, C.-S. Lin, S.-S. Chang, and P.-A. Hsiung. Dynamically swappable hardware design in partially reconfigurable systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2742–2745, May 2007.

[6] E. Schaefer. A simplified data encryption standard algorithm. *Cryptologia*, 20:77–84, January 1996.

[7] B. William and L. Joan. *JPEG: Still Image Data Compression Standard.* Kluwer Academic Publisher, 1993.