RECONFIGURABLE HARDWARE MODULE SEQUENCER FOR DYNAMICALLY PARTIALLY RECONFIGURABLE SYSTEMS

Kai-Jung Shih, Chin-Chieh Hung, Pao-Ann Hsiung

Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi, Taiwan 621, ROC.

ABSTRACT

Traditionally, 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 among computing units, but also adopts a programming model similar to that of 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 not only superior to the networked architecture in terms of lower communication load and to the data flow architecture in terms of reduced system complexity, but also performs better than a similar OPB-Dock based architecture.

Key words: reconfigurable module sequencer (RMS), dynamically partially reconfigurable system (DPRS), processor-controlled networked architecture (PNA), data sequencer-controlled networked architecture (SDA), memory mapping.

I. INTRODUCTION

The term "reconfigurable computing" was first proposed by Estrin [7] in 1960 and popularized by the FPGA technology in the mid-1980s. As a tradeoff between general-purpose computing and application-specific integrated circuit (ASIC) computing, reconfigurable computing has combined the advantages of the two extreme characteristics [18], [19]. The performance of reconfigurable systems is better than general-purpose systems and the cost is smaller than that of ASICs. The main advantage of a reconfigurable system is its high flexibility and the design effort is between that of general-purpose processor and ASICs.

Unlike von-Neumann based architectures, there are currently no standard memory hierarchy and communication schemes for *dynamically partially reconfigurable system* (DPRS) [6], [8], [11], [12], [13], [16], [20]. However, two communication architectures are commonly adopted, namely *processor-controlled network architecture* (PNA) and *sequencer-controlled data flow architecture* (SDA). In PNA, the processor is responsible for all data transfers between hardware and software or between two hardware modules. As an application is divided into a large number of small reconfigurable hardware blocks, the overhead for the processor in handling communication between these small hardware blocks increases rapidly. In SDA, data communication is completely handled by a devoted data sequencer, which requires the manual or automatic generation of very low level data movement instructions. There is no processor in SDA.

Both the PNA and the SDA architectures pose limitations for a system design to achieve high communication performance. In PNA, the processor is easily overloaded with too many communication requests, as a result of which, the overall system performance is degraded. In SDA, the main problem 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 such as PNA and the high complexity of data flow architectures such as SDA, a novel *Module Sequencer Architecture* (MSA) is proposed in this work, which solves all the above four is-

Manuscript received Jan. 7, 2008; revised Mar. 3, 2008, and accepted Mar. 14, 2008.

sues. The concept of such a sequencer has been proposed in the cryptography processor [10] using "microcode" as the instruction sequence.

The contributions of this work are as follows.

- *Reduced communication overhead*: In contrast to PNA, in MSA the module sequencer shares a significant portion of data communication workload, thus easing the burden of the processor, the DMA, and the OS. The module sequencer controls function blocks to directly communicate with each other.
- *Simplified programming model*: In contrast to SDA, a much simpler programming model is adopted in MSA, which is coherent with the conventional processor/FPGA architecture.
- *Simplified bus architecture*: Instead of a complex full function bus, a simple read-write control signal driven bus architecture is proposed for interconnecting the swappable hardware function blocks.
- *Virtual function mapping*: Hardware function blocks can be dynamically relocated into different positions in the reconfigurable logic with a mapping between logical task ID and physical slot ID, while maintaining all communications.

This article is organized as follows. Section II discusses related research work and compares them with our architecture. The proposed module sequencer architecture is described in Section III. The illustration examples are given in Section IV. Finally, conclusions are given in Section V along with future work.

II. RELATED WORK

Several architectures [1], [2], [3], [5], [8], [16], [17], [20] have been designed for realizing a DPRS, however different architectures cause varying degrees of impact on the communication overhead for a processor. In this section, we focus on two typical PNA and two SDA for DPRS.

A DPRS architecture with OS frames [17], [20] is proposed to alleviate the communication problems with a bus. Hardware tasks communicate with each other by the unified interface that is the bus. In this architecture, two OS frames which bridge the static and dynamic modules are located at the left and right edges of the FPGA. OS frames are bus arbiters which control the bus and allocate its usage to requesting tasks. The static and dynamic zones are separated by this approach. The OS frame left contains the Bus Arbiter Left (BARL) that controls the left bus which sends data from right to left. The OS frame right contains the Bus Arbiter Right (BARR) that has the same capability in the opposite data transfer direction. A reconfigurable hardware task includes a bus access controller (BAC) to connect to the bus. A BAC has a task control interface (TCIF) and m data exchange interfaces (DxIF). The TCIF handles the bus protocol and the DxIF controls the sent and received data. When a system is under reconfiguration, the arbiters freeze the bus because reconfiguration in Xilinx Virtex-II is column-wise which breaks the bus. Beside the arbiters, an operating system for reconfigurable systems (OS4RS) [4] is needed to enforce correct communication between the underlying reconfigurable hardwares and therefore the processor takes a lot of time for communication.

Ferreira proposed an OPB DOCK [8], [16] to integrate the original bus architecture of a system, for example IBM CoreConnect, and a dynamically partially reconfigurable subsystem using the Xilinx bus macro. This approach also uses a bus as the communicating medium. The processor communicates with other hardware components via the CoreConnect bus, which is composed of the Processor Local Bus (PLB), the On-chip Peripheral Bus (OPB), and the OPB DOCK architecture. The PowerPC accesses the BRAM via the PLB, the UART via the OPB, and the reconfigurable area via the OPB DOCK. Such a system has the advantages of a common bus architecture and also works perfectly with DPRS. This approach takes a lot of processor cycles to control and synchronize the communication between reconfigurable hardware function blocks because the OPB DOCK is like an arbiter or bridge between the OPB bus and the reconfigurable area without any advanced communication control. Our work will be based on a similar architecture.

Transport Triggered Architecture (TTA) [1], [2], a move machine, was proposed for customizing application-specific instruction-set processor (ASIP) designs. The TTA contains a set of buses with sockets that can be used to connect small hardware circuits such as adders, subtractors, registers, and so on. Instruction execution is triggered by data movements between different hardware circuits, and the real execution of the hardware is triggered by the side effect of the data movement when this movement target is a trigger register. 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], [5] is a domain-specific coarse-grained reconfigurable architecture with a stream manager, an instruction generator, a configurable instruction decoder, a configurable interconnect, and application-specific function units such as ALU, multiplexer, RAM, and registers. Data I/O is performed by the stream manager and dynamic configurations of function units and routing are performed by instructions that are executed by cycle-by-cycle sequencing. RaPiD is a typical data flow architecture with a data sequencer.

Both the TTA and RaPiD architectures work as superscalar processors to speedup specific applications. However, since the architectures deviate significantly from the conventional processor/FPGA architecture, their programming models are very complex. Our proposed module sequencer architecture is not only compatible with the conventional architecture, but also allows a simpler programming model.

III. MODULE SEQUENCER ARCHITECTURE

Similar to other DPRS architectures, the target module sequencer architecture has a statically configured part and a dynamically reconfigurable part. As illustrated 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 and consists of a data bus, a set of read-write control signals, and a reconfigurable area that can be configured with hardware function blocks along the area width.

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. The microprocessor is responsible only for coarse-grained communication and control, while the module sequencer is responsible for fine-grained communication and control. The RMS in MSA can be regarded as a reconfigurable instruction co-processor, with a much coarser instruction granularity (e.g., DCT, Quantization) than that of traditional instructions (e.g., add, sub, jump). Thus, MSA achieves reduced communication load for the processor and also adopts a programming model that is coherent with the conventional processor-FPGA model and is simpler than that in SDA.

The RMS communicates with the static hardware part through a static memory mapping and executes commands sent from the processor by controlling hardware function blocks that are configured into the reconfigurable area. 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 ID mapping is illustrated in Figure 1. The same logical ID function can be mapped to different physical ID slots. When an application requests a function with a specific logical ID, the RMS maps this request to the physical ID of an idle slot configured with that function. For example, a quantization function request can be mapped by RMS to either slot 1 or slot 4, also shown in Figure 1.

Data pipelining between hardware function blocks, used for executing a command, is completely controlled by the RMS through the data and control bus in the reconfigurable area. For example, in a JPEG encoder [21], the data is pipelined from a *DCT* block to a *Q* block and then from the *Quantization* (*Q*) block to an *Entropy Encoding* (*EE*) block by the RMS, where the processor command is a function sequence called a chain and is denoted by <*SW*, *DCT*, *Q*, *EE*, *SW*>, where *SW* represents corresponding software.

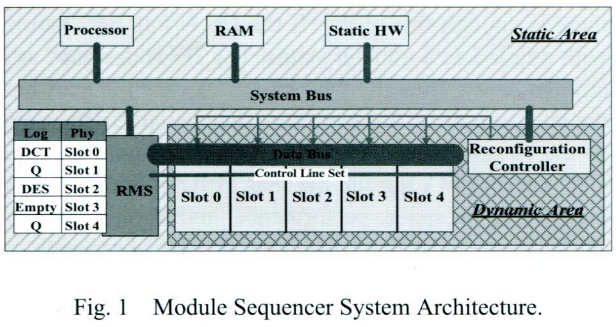


Fig. 1 Module Sequencer System Architecture.

A. Communication Flow

A basic principle of MSA is that all communication links are dynamically configured. The source and destination blocks in a data transfer do not know in which slot the other block is located physically. For example in the JPEG encoder sequence, the link between *DCT* and *EE* is configured only after the DCT data are computed. *DCT* also does not know to which *EE* block are the computed data sent. This scheme allows the RMS to concurrently and efficiently run several applications within the reconfigurable area. In MSA, an application is defined by a set of partially ordered command sequences called chains. Each chain $< f_0, f_1, \cdots, f_n, f_{n+1} >$ consists of a sequence of $n + 2$ functions, where f_n, f_{n+1} are software functions and $f_1, \cdots, f_n$ are hardware functions. A data transfer request is defined by a pair of adjacent functions $< f_{i-1}, f_i >$. The processor and the RMS share the communication workload as follows.

1. The software application running on the processor sends a chain command to the RMS through a system call interface.
2. The RMS driver in the OS sends the input data for that chain to the RMS.
3. The RMS manages the communication for each data transfer request in the chain.
4. When the RMS finishes the chain execution, it interrupts the processor to notify it to retrieve the chain's output data buffered in the RMS.

B. Reconfigurable Module Sequencer 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 logical ID to physical ID mapping, the usage status (reset or configured), and the execution status (idle or running) if it is configured.

To manage the three internal storages, RMS has a command pool controller (CPC) that accesses the CP, stores chains into it, and selects an enabled data transfer request to be executed from some chain. RMS also has a memory controller (MC) that loads input data from the DF to the configurable data bus and a slot controller (SC) that accesses ST and controls the reconfigurable bus by asserting and deasserting the control signals. For output data management, RMS has an 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.

C. RMS Bus Interface Design

Each slot has a dedicated read, write, and request signal, whereas the busy signal is shared by all slots. This unified interface is illustrated in Figure 3. It must be noted here that we do not make any assumptions on the size of the slot or their relative physical location in the reconfigurable area, which means that the OS can decide to configure any portion of the reconfigurable area as any slot as long as the slot configuration connects to the dedicated and shared control signals. Reconfigurable function blocks are dynamically relocated by the placer in the OS4RS to achieve this location independence

D. RMS Controllers Design

The RMS input is classified into three types, which are the mapping between logical ID and physical ID, chain commands, and chain data, which are handled by different controllers. As we mentioned above, the RMS has four controllers, one decoder, and one monitor to manage these data.

The command pool controller (CPC) is responsible for storing chain commands, for maintaining the state of

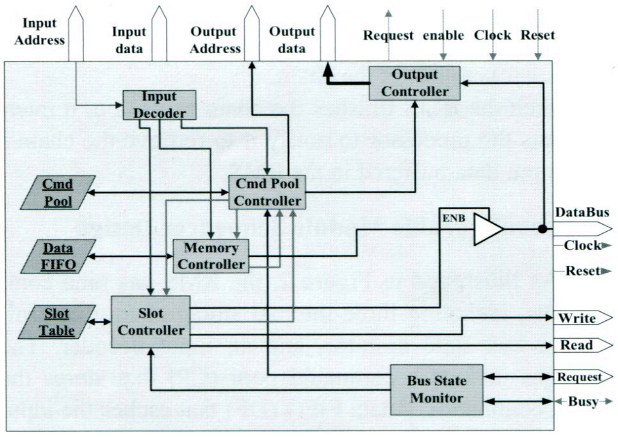


Fig. 2 Reconfigurable Module Sequencer.

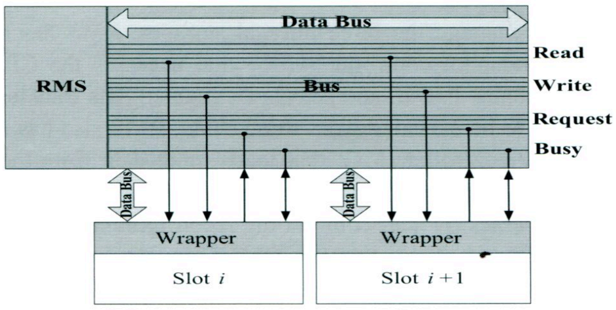


Fig. 3 Reconfigurable Bus Interface.

chain functions as running, waiting, transferring, or idle, and for deleting a chain. The CPC is the controller of a chain, which dispatches commands, notifies other controllers, and selects a data transfer request. The CPC will check a data transfer function pair $<f_i, f_{i+1}>$ in a chain function by function. For the first request pair, the CPC notifies the MC to check if the input data for this chain is available. For the last request pair, the CPC notifies the OC to latch the data on the reconfigurable bus and the OC will interrupt the processor to retrieve the output data. Then the CPC will check the responses from the MC and the SC. If the responses are positive, the CPC initiates the data transfer, otherwise the CPC will check another data transfer request. We use the chain priority to select the data transfer function, which causes some issues such as priority inversion and starvation. Nevertheless, the handling of these issues is left to the operating system so that the RMS is small, compact, and platform independent.

Because hardware functions can be swapped in and out from the reconfigurable area, the static placement of reconfigurable hardware function blocks will pose severe limitations on system flexibility. To deal with this issue, logical IDs are used to request specific functions, and the RMS translates the logical ID to a physical ID dynamically. The logical ID and physical ID mappings are stored in the Slot Table (ST). The SC modifies slot states, controls the bus signals for communication, and makes sure the quiescent state is reached before the reconfiguring process. When the CPC checks if the data transfer between a pair of functions is possible, the SC searches for a slot configured with the function block having the same logical ID as that of the destination function. If such a function exists, the SC returns the slots with functions that could be assigned. If no such function exists, the SC sends a hardware page fault signal to the OS4RS. Hardware page faults can be used as an indicator for temporal locality by the placer and the scheduler. The placer and the scheduler can either swap-in the requested hardware module immediately or suspend the chain by delaying the swap-in. When the CPC makes sure all resources are available, it drives the permit signal to the SC which starts the data transfer between the pair of enabled functions.

The MC caches the chain data sent by the OS4RS and sends the data to the data bus as needed. This controller is active only when performing the first data transfer in a chain. When the chain data is sent into the RMS by the OS4RS, the MC caches the data in DF if there is enough free space. If the DF is full, MC returns with a failed data transfer status. The DF is designed as a circular array, thus the MC has to modify the start and the ending indices that represent the available data range of the circular array. The MC is much simpler than the CPC and the SC, but the MC has different design trade-off choices on space, performance, and implementation. These issues have been discussed in a lot of research on memory controllers [15].

The input decoder and output controller are used to handle the input from the OS4RS to the RMS and the output from the RMS to the processor, respectively. The input decoder checks the input command types, such as chain command, chain data, and slot mapping messages, and dispatches these commands and data to the CPC, SC, or MC. The input decoder verifies the input type as either logical and physical ID mapping message, chain commands, or chain data and then asserts the corresponding controller to act accordingly.

The OC is active when a chain finishes execution. It latches the RMS bus data, interrupts the processor to notify the chain is finished and prepares the output data.

The Bus State Monitor (BSM) handles the control signal interface between the RMS and reconfigurable area bus. The BSM monitors the bus states, controls the bus signals such as busy or request signal, and relays the bus information to the SC and the CPC. When the busy signal is asserted by a reconfigurable hardware function, the BSM notifies the SC to modify this function state as running and notifies the CPC to continue selecting another function to execute. Then the BSM deasserts the busy signal for the next data transfer. The BSM monitors the reconfigurable area bus states and notifies the controllers about the bus states. The design of a dedicated bus state monitor can reduce the complexity of the RMS. If all the bus state signals are connected to controllers directly, the signals such as busy signal deassert is controlled by the controller itself. If more than one controller connects to the same signal, the complexity of signal control and synchronization will increase exponentially.

E. RMS Control Flow

We will now describe the interaction between the RMS and the processor. We use the chain $<f_0, f_1, f_2, f_3, f_4>$ as an example, where f_0, f_4 are software, and f_1, f_2, f_3, are reconfigurable hardware modules. The processor sends three types of commands to the RMS in sequential order that include logical to physical slot ID mappings, chain commands, and input data for each chain. The ID mappings are stored in the ST by the SC, the chain commands are stored in the CP by the CPC, and the input data are stored in the DF by the MC. The CPC checks for data transfer requests in the CP and selects a request belonging to the chain with the 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 the MC to check if its input data are available in the DF if it is the first data transfer request, $< f_0, f_1>$. 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 (have a logical to physical ID mapping). If configured and unused, the 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. For example, for DCT to send data to quantization, the SC asserts the DCT write signal and the quantization read signal, then the DCT puts data on the reconfigurable data bus and the quantization receives the data. If the sender is a SW, the SC enables the tri-state buffer as illustrated in Figure 2, so that the data of the chain sent by the MC can be transferred on the bus. If it is not configured or all physical instances are busy, then the SC issues a hardware page fault to the microprocessor and acknowledges that the requested function is unavailable. In this case, the operating system invokes the placer and the scheduler to configure a corresponding hardware function into the reconfigurable area using the configuration device.

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.

After a hardware function completes computation, it asserts the request signal to request a data transfer such that the successor function is enabled after the data transfer . The bus state monitor helps accomplish the data transfer.

F. Programming Model

Any new architecture should have an associated 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 illustrated in Figure 4, given a user program, for example a C program, and corresponding task profile information, the chain generator determines the hardware-software partition, that is, which computation-intensive loops must be implemented as reconfigurable hardware chains, and the rest of the program as software. 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, a *Chained Program Operating System* (CPOS) is designed. Besides being an OS4RS, the CPOS has a

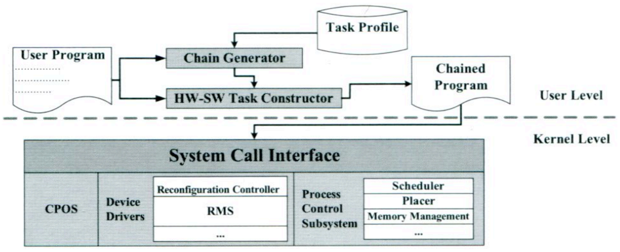


Fig. 4 Programming Model and Chained Program Operating System.

system call that allows chained programs to send a request for executing a chain through the RMS driver. The 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 the 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. Details on how RMS knows which blocks are already configured (placed) and how to request a new block configuration were discussed earlier in section III.D. The development of CPOS is still an on-going work and requires further design and implementation.

G. 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 $< f_0, f_1, \cdots, f_n, f_{n+1} >$, 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,
- T_{DMA}: DMA setup time in cycles,
- $SZ_{DF}DFS$: RMS Data FIFO size in bytes, and
- SZ_{IN}: 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: The processor controls the complete data transfer by itself, thus the total time, in cycle counts, is the sum of the time for each data transfer request multiplied by the total number of iterations k, as given in Equation (1).

$$k \times \sum_{i=0}^{n} t_i \qquad (1)$$

- No RMS, With DMA: The processor sets the DMA to transfer data, thus the total time, in cycle counts, is $k \times (n+1)$ times the DMA setup time, as given in Equation (2).

$$T_{DMA} \times k \times (n+1) \qquad (2)$$

- With RMS, No DMA: The processor sends the chain command and the input data for the chain to RMS, and then the RMS controls the data transfers between the functions, without the processor. The commands take $(n+2)$ cycles, and the total time for data transfer is $(t_0 + t_n) \times k$ cycles. Totally, the cycle counts are as given in Equation (3).

$$(t_0 + t_n) \times k + (n+2) \qquad (3)$$

- With both RMS and DMA: The processor sets the DMA to transfer data to RMS. The chain command transfer takes (n + 2) cycles, and the number of times DMA must be setup is SZ / DFS . Totally, the cycle counts are as given in Equation (4).

$$T_{DMA} \times \lceil SZ_{IN} / SZ_{DF} \rceil \times 2 + (n+2) \qquad (4)$$

Since the OPB-Dock based Architecture (ODA) proposed by Ferreira and Silva [8], [16] is similar to our architecture, we model the performance of their architecture as Equation (5) and perform the simulation in Section IV for this performance model.

$$(t_0 + t_n) \times k + T_{PS}(n-1) \qquad (5)$$

In the ODA architecture, the processor sends the input data to reconfigurable hardware through OPB-dock. Suppose the processor synchronization between two functions takes TPS cycles, thus totally it takes T_{PS} $(n-1)$ cycles for the n hardware functions. The total time for data transfer between the OS4RS and the first and last hardware functions is $(t_0 + t_n) \times k$ cycles.

H. Preemption Issue

In the RMS system, a chain contains one or more functions and can be preempted only between two adjacent functions. Since a function is implemented with hardware IP, which basically means that reconfigurable hardware components are non-preemptive. If a low priority chain has a large function that executes for long time durations and if there is an urgent high priority chain to be executed, we observe the convoy effect. Due to the lack of reconfigurable resources, the high priority chain will have to wait for the low priority large function to finish execution.

To alleviate this problem, a wrapper has been proposed and designed for hardware task preemption [9]. The integration of RMS and this hardware task wrapper will be performed in the CPOS in the future and is out of scope here.

IV. EXPERIMENTS

The target module sequencer architecture was modeled, designed, and implemented. However, for performance evaluation we developed a System C-based simulation framework for the proposed architecture, which consisted of a processor, RAM, an optional DMA, a reconfigurable area with RMS, a system bus, and a reconfigurable bus. Both busses are 32 bits wide, the RMS cache size is 4096 byte, and the DMA setup time is 15 cycles. For the OPB-Dock based Architecture, we assume the processor synchronization time between two tasks T_{PS} is 6 cycles. We first experiment with three applications that have a single chain each. Then, we experiment with multiple chains. The first example was used for checking feasibility of the proposed architecture. It has three reconfigurable hardware functions f_1, f_2, f_3. 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. The chain command is $< f_0, f_1, f_2, f_3, f_4 >$, where f_0, f_4 are software functions.

The second example is a subsequence of the Joint Photographic Experts Group (JPEG) encoder. As illustrated in Figure 5, a JPEG encoding task is composed of six functions, namely hue transformation, sampling, discrete cosine transform (DCT), quantization, entropy encoding, and header combination. The hue transformation function transforms an R-G-B image to a Y-Cb-Cr image. The sampling function processes the hue-transformed image block by block and transforms each block from 16 x 16 pixels to 8 x 8 pixels. Users can choose the 411 method or the 211 method according to the image quality required. The DCT function is the discrete cosine transform, that transforms an image from the spatial domain to the frequency domain. The quantization function uses a quantization table for a DCT transformed image and results in less entropy for compression. The entropy encoding function processes the result data from the quantization function by performing DC entropy encoding and AC entropy encoding. The header combination integrates the results from the entropy encoding with the information of image size, sampling method, and quantization table. Since the function sequence from DCT to Entropy Encoding is a computation-intensive loop and can thus be partitioned into a chain of reconfigurable hardware. The other functions including hue transformation, sampling, and header combination will be executed as software on the microprocessor. The JPEG encoder functions can be formed as *<SW, DCT, Q, EE, SW>*. We assume the example input is an 800 x 600 pixels gray-level image, thus the total input data size is 480,000 bytes. The sampling function is neglected in our example so that the input data to DCT function are 480,000 pixels and the number of iterations from DCT function to Entropy Encoding is 7,500.

The third example is a Data Encryption Standard (DES) [14] block that takes 64-bit data as information and 64-bit data as key. The total data for this example is 1024 bytes for data and 1024 bytes for key. We assume a single execution takes 18 cycles to encrypt 64-bit data. The chain to be executed is *<SW, DES, SW>*.

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. From Table 1, we can observe that the performance improvement (6%) is limited when we use only DMA, whereas the RMS alone brings very good performance improvements (70%), especially when the chain is time-consuming such as the JPEG encoder. 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 DES example has only one function with no chance of load reduction while still requiring additional RMS setup time. The combination of DMA and RMS results in the best performance improvements (96%~99%) in all cases.

The reason that the RMS/DMA combination provides very large processor load reduction is because without RMS, a single DMA setup can transfer only a limited amount of data, equivalent to that consumed by one iteration of a chain, 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, which if large enough could accommodate all the data required by all the k iterations of a chain. This performance improvement is approximately equivalent to that provided by SDA, which also almost totally relieves the processor of all communication loads.

Comparing the performance of ODA and MSA using Equations (3) and (5), we can see that although both of them result in a decrease in communication load for the processor, MSA still outperforms ODA. In MSA, the RMS takes care of all synchronizations between reconfigurable function blocks and thus alleviates the processor of almost all synchronizations among hardware blocks. In ODA, the processor still needs to handle the synchronizations between two functions in reconfigure-

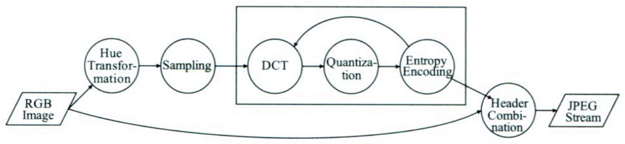


Fig. 5 JPEG Encoder Flow.

Table 1 Processor Cycles for Handling Communication.

Performance Model	RMS	DMA	Toy	JPEG	DES
Eq. (1)	No	No	2,048	510,000	1,024
Eq. (2)	No	Yes	1,920 −6.25%	450,000 −1.18%	960 −6.25%
Eq. (3)	Yes	No	1,029 −49.76%	150,005 −70.59%	1027 +0.29%
Eq. (4)	Yes	Yes	35 −98.29%	3,545 −99.31%	35 −96.58%
Eq. (5)	OPB Dock	No	1,042 −49.12%	150,012 −70.59%	1,024 -0%

able hardware, hence that the OPB dock in the ODA relieves the processor of only a small portion of communication synchronizations. If a chain has only one function configured into reconfigurable hardware, the ODA gives a better result than the proposed MSA architecture because of the overhead caused by RMS in MSA, while the OPB dock does not cause that much overhead. Hence, MSA needs more than one function in a chain to amortize the RMS overheads. This is also quite acceptable because no one would like to only accelerate a single hardware function with RMS.

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.

The single experiment described above might not provide enough basis for consolidating our claim on improving performance in RMS by assigning a higher priority to the time-consuming chain. Hence, we performed 18 different experiments, in two groups of 9 each, as illustrated in Table 3. The application consisted of two chains $C_1 =< f_0, f_1, f_2, f_3, f_4 >$ and $C_2 =< f_0', f_1', f_2', f_3' >$. Keeping the total input data size (SZ_{C2}) fixed at 2048 bytes for C_2, we varied the total input data size (SZ_{C1}) of C_1 from 256 bytes to 2304 bytes, in increments of 256 bytes, for each of the 9 experiments in each group. For the first group S, C_1 was assigned a higher priority than C_2 (denoted by $\pi_{C1} > \pi_{C2}$), while for the second group S', C_2 had a higher priority than C_1 (denoted by $\pi_{C1} < \pi_{C2}$). The variation in total input data size for C_1 was performed so that we could observe the cases where C_1 was less time- consuming, that is, $T_{C1} < T_{C2}$ (as in experiments (1) to (7)) and also the cases where C_1 was more time-consuming, that is, $T_{C1} > T_{C2}$, (as in experiments (8) and (9)) when compared to C_2. We can observe from Table 3 that by assigning a higher priority to the more time-consuming chain C_2 in the first 7 experiments, the total time for executing the two chains concurrently is less than that under an inverse priority assignment. For experiments (8) and (9), because the total input data of C_1 has increased resulting in its total execution time exceeding that of C_2, which is T_{C2} = 4563 cycles. Now, it turns out that assigning the more time-consuming chain C_1 a higher priority results in a shorter system execution time (marked in bold in Table 3).

From the above experiments, we can conclude that in general assigning the time-consuming chain a higher priority results in a shorter system execution time. This policy of priority assignment is implemented into the

Table 2 Execution Time of JPEG and DES with Different Priorities.

#	JPEG	DES	JPEG Time	DES Time	System Time
1	Yes	No	462	N/A	462
2	No	Yes	N/A	526	526
3	Yes	Yes	462	526	**988**
4	Low	High	558	702	**702**
5	High	Low	525	953	**953**

Low and High are priorities of JPEG and DES

Table 3 Performance Comparisons under Different Priority Assignments.

#	SZ_{C1} (bytes)	T_{C1} (cycles)	$\pi_{C1} > \pi_{C2}$			$\pi_{C1} < \pi_{C2}$		
			T_S	R_S	$R_S\%$	$T_{S'}$	$R_{S'}$	$R_{S'}\%$
1	256	606	4781	388	7.5%	**4659**	510	9.9%
2	512	1198	5005	756	13.1%	**4755**	1006	17.5%
3	768	1790	5229	1124	17.7%	**4851**	1502	23.6%
4	1024	2382	5453	1492	21.5%	**4947**	1998	28.8%
5	1280	2797	5677	1826	24.7%	**5043**	2494	33.1%
6	1536	3566	5901	2228	27.4%	**5459**	2670	32.9%
7	1792	4158	6125	2596	29.8%	**6051**	2670	30.6%
8	2048	4750	**6349**	2964	31.8%	6643	2670	28.7%
9	2304	5342	**6941**	2964	29.9%	7235	2670	27.0%

SZ_{C2} = 2048 bytes, T_{C2} = 4563 cycles

π_{Ci} : Priority of C_i (larger value means higher priority)

32 bytes are consumed per iteration for both chains

the number of iterations $k_{Ci} = SZ_{Ci} / 32$

$R_S = T_{C1} + T_{C2} - T_S$, $R_S\% = R_S/(T_{C1} + T_{C2})$, $R_{S'} = T_{C1} + T_{C2} - T_{S'}$

$R_{S'}\% = R_{S'} / (T_{C1} + T_{C2})$

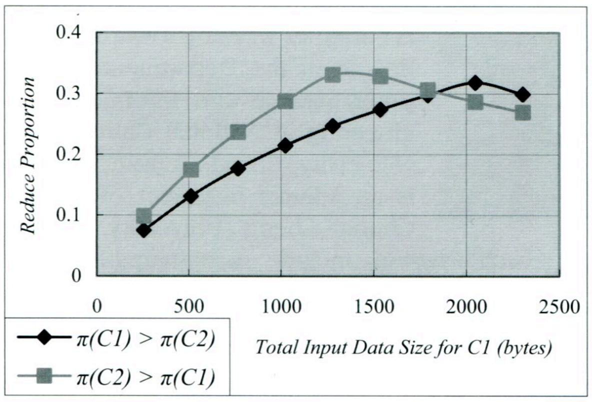


Fig. 6 Priority Assignment Policies.

RMS driver design for performance enhancement in RMS-controlled MSA systems. Figure 6 illustrates the relationship between the total input data size SZC1 and the reduction proportions R_S and $R_{S'}$ when $\pi_{C_1} > \pi_{C_2}$ and $\pi_{C_1} < \pi_{C_2}$, respectively. The RMS driver can use the data size as a reference for deciding which chain to assign a higher priority because the chain execution time T_{Ci} generally increases with data size SZ_{Ci}.

As far as comparison with SDA is concerned, it is difficult to measure the design complexity of SDA. As an empirical analysis, we can compare the performance model of MSA with that of SDA. Since SDA has no operating system, users need to implement their applications from a behaviour level rightdown to register-transfer level (RTL) by hand. Users who want to use SDA need to know the data sequencer behaviour and adopt it for their application, which is quite difficult for a normal user. Further if the application problems that users want to solve are too complex, the SDA also results in poor scalability. Further, the verification for SDA architecture is difficult because the behaviour is hard to model.

V. CONCLUSION

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. A simple programming model was also proposed, which is coherent with the conventional processor-FPGA system architecture. Experiments were presented to validate the feasibility and benefits of this architecture. A priority-based scheme that resulted in shorter schedules was also presented with experiments. Future work will consist of support for preemptive hardware functions and the integration of RMS with the scheduler and placer in the CPOS.

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*, pp. 130-135, IEEE Computer Society, March 1994.

[2] H. Corporaal and H. Mulder “Move: A framework for high-performance processor design,” In *Proceedings of the IEEE Conference on Supercomputing*, pp. 692-701, IEEE Computer Society, 1991.

[3] D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling “Architecture design of reconfigurable pipelined datapaths,” In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, pp. 23-40, IEEE Computer Society, March 1999.

[4] S. -H. Chang and P. -A. Hsiung, “Operating System for Reconfigurable Systems,” *Electron Technology Information Magazine*, Vol. 64, pp. 20-26, Gallant Company, April 2006.

[5] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S.G. Berg, “Mapping applications to the RaPiD configurable architecture,” In *Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, pp. 106-115, IEEE Computer Society, April 1997.

[6] M. Edwards and P. Green, “Run-time support for dynamically reconfigurable computing systems,” *Journal of Systems Architecture*, Vol. 49, No. 4-6, pp. 267–281, September 2003.

[7] G. Estrin, “Organization of Computer Systems: The Fixed Plus Variable Structure Computer,” In *Proceedings of the Western Joint Computer Conference*, San Francisco, USA, pp. 33-40, May 1960.

[8] J. C. Ferreira and M. M. Silva, “Run-Time Reconfiguration Support for FPGAs with Embedded CPUs: The Hardware Layer,” In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Vol. 4, pp. 165a, IEEE Computer Society, April 2005.

[9] 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), New Orleans, USA, pp. 2742-2745, IEEE Computer Society, May 2007.

[10] J. Goodman , A. P. Chandrakasan, “An energy-efficient reconfigurable public-key cryptography processor,” *IEEE Journal of Solid-State Circuits*, Vol. 36, No. 11, pp. 1808-1820, November 2001.

[11] Y. E. Krasteva, E. de la Torre, and T. Riesgo, “Par-

tial reconfiguration for core reallocation and flexible communications," In *Proceedings of the 2nd International Workshop on Reconfigurable Communication-Centrics Systems-on-Chip*, pp. 91-7. IEEE Computer Society, July 2006.

[12] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, "Interconnection networks enable fine-grain dynamic multitasking on FPGAs," In *Proceedings of the 10th International Workshop on Field Programmable Gate Arrays (FPL)*, pp. 795-805, Springer Verlag, September 2002.

[13] T. Marescaux, V. Nollet, J. -Y. Mignolet, A. Bartic, W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Run-time support for heterogeneous multitasking on reconfigurable SoCs," *VLSI Journal of Integration*, Vol. 38, No. 1, pp. 107-130, Elsevier Science, October 2004.

[14] E. Schaefer, "A Simplified Data Encryption Standard Algorithm," *Cryptologia*, Vol. 20, No. 1, pp. 77-84, January 1996.

[15] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, sixth edition, John Wiley, June 2001.

[16] M. L. Silva and J. C. Ferreira, "Using a Tightly-Coupled Pipeline in Dynamically Reconfigurable Platform FPGAs," In *Proceedings of the 8th Euromicro Conference on Digital System Design (DSD)*, pp. 383-387. IEEE Computer Society, August September 2005.

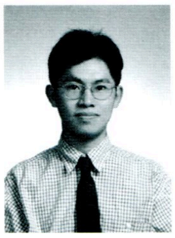
[17] S. Steinegger, *Reconfigurable Hardware OS Prototyp-Part FPGA*, Master's thesis, Eidgenossische Technische Hochschule (ETH), Zurich, May 2004.

[18] R. Tessier and W. Burleson, "Reconfigurable Computing for Digital Signal Processing: A Survey," *Journal of VLSI Signal Processing*, Vol. 28, No. 1-2, pp. 7-27, 2001.

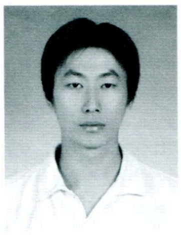
[19] N. Tredennick, "The Case for Reconfigurable Computing," *Microprocessor Report*, Vol. 10, No. 10, pp. 25-27, 1996.

[20] H. Walder and M. Platzner, "Implementation of a Runtime Environment for Reconfigurable Hardware Operating Systems," *Technical Report TIK Nr. 195*, Swiss Federal Institute of Technology (ETH), Zurich, June 2004.

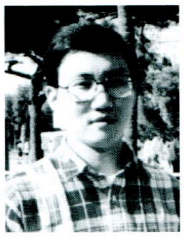
[21] B. P. William, and L. M. Joan, *JPEG: Still Image Data Compression Standard*, Kluwer Academic Publisher, ISBN: 0442012721.



Kai-Jung Shih received his M.B.A. degree in Management Information Systems at National pingtung University of Science and Technology, Pingtung, Taiwan, ROC, in 2000. He is currently working towards a Ph.D. degree in the Department of Computer Science and Information Engineering at National Chung Cheng University, Chiayi, Taiwan, ROC. He is an adjunct instructor of National Chung Cheng University. His research interests include reconfigurable system design, embedded systems design and hardware/software co-design.



Chin-Chieh Hung received both B.S. and M.S. degrees in Computer Science and Information Engineering at National Chung Cheng University, Chiayi, Taiwan, ROC, in 2005 and 2007, respectively. From 2005 to 2007, he was a research assistant in the Department of Computer Science and Information Engineering at National Chung Cheng University. Currently, he is a system software engineer at Realtek Semiconductor Corporation, Hsinchu, Taiwan, ROC. His research interests include reconfigurable systems design, embedded systems design and operating system.



Pao-Ann Hsiung received his B.S. degree in Mathematics and a Ph.D. degree in Electrical Engineering from National Taiwan University, Taipei, Taiwan, ROC, in 1991 and 1996, respectively. From 1996 to 2000, he was a post-doctoral researcher at the Academia Sinica, Taipei. From February 2001 to July 2002, he was an assistant professor and an associate professor from August 2002 to July 2007 in the Department of Computer Science and Information Engineering, National Chung Cheng University (CCU), Taiwan. He is currently a full professor. He received the Young Scholar Award from CCU in 2004. He is a senior member of the IEEE, a senior member of the ACM, and a life member of IICM. He has been listed in several well-known professional biography listings. He has published more than 140 papers in international journals and conferences on various topics including reconfigurable computing, real-time embedded systems, and formal verification. He is currently an associate editor of several journals including IJES, IJMUE, JSE, OSE, and IJOP and is on the technical program committee of several international conferences.