EFFICIENT EXTERNAL MEMORY INTERFACE FOR MULTI-PROCESSOR PLATFORMS REALIZED ON FPGA CHIPS

Hristo Nikolov *Todor Stefanov* *Ed Deprettere*

LIACS, Leiden University, The Netherlands
email: {nikolov, stefanov, edd}@liacs.nl

ABSTRACT

The complexity of today's embedded applications requires modern high-performance embedded System-on-Chip (SoC) platforms to be multiprocessor architectures. Advances in FPGA technology make the implementation of such architectures in a single chip (MPSoC) feasible and very appealing. In recent years, the FPGA vendors integrated enormous amount of hardware resources in their FPGAs allowing larger and more complex MPSoCs to be built in their FPGA fabric. The main limitation on the size of an MPSoC that can be built in a single FPGA appears to be the amount of on-chip memory. To relax this limitation, the usage of external (off-chip) memory has to be considered. The state-of-the-art development tools support off-chip memory for (multi-master) shared bus architectures with arbitration of the memory accesses. Such architectures might be efficient for single processor systems however for multiprocessor systems the shared bus concept significantly limits the systems performance even if a DMA mechanism is used.

In this paper we present our approach and interface when using an external memory for inter-processor data communication in multiprocessor platforms. We propose a hierarchical memory system with a programmable controller to transfer data between external and on-chip memories using a DMA mechanism. Our approach does not require arbitration which results in better overall performance. Results demonstrating the effectiveness of the proposed hierarchical memory system are presented as well.

1. INTRODUCTION

The complexity of embedded multimedia and signal processing applications has reached a point where the performance requirements of these applications can no longer be supported by embedded system platforms based on a single processor. Therefore, modern embedded system platforms have to be multiprocessor architectures. Although today's FPGA technology allows for building such architectures due to the large amount of resources integrated in a single FPGA, still the amount of integrated on-chip memory is a severe limiting factor for building large and high performance multiprocessor systems (MPSoCs) on an FPGA. This is because stream-based computations, common in multimedia and signal processing applications typically manipulate large volumes of fine-grain data that, despite recent increases in FPGA memory capacity, must be mapped to external memories and then streamed through the FPGA for processing.

Commercially available tools [1][2] provided by leading FPGA companies (e.g., Xilinx and Altera) offer limited capabilities (approaches, interfaces, and components) to deal efficiently with external (off-chip) memories. The current state-of-the-art tools support external memory access based on a shared multi-master bus architecture with arbitration. Such memory access is very inefficient when a high performance multiprocessor system (MPSoC) on an FPGA chip is considered. Moreover, the shared bus arbitration guarantees only that a processor can physically access the external memory resources connected to the bus. However, it can not schedule and synchronize an application dependent accesses of different processors so that it may fail the correct functional behavior of the application and to protect from incorrect accesses of one processor to the external memory space of another processor. The problem of dealing with an external memory is exacerbated by the lack of automated programming approaches for MPSoCs on FPGAs with an external memory. In addition, the diversity of vendor-specific external memory interfaces results in a very time consuming manual effort for designers to match the MPSoC design specification to a specific interface which makes the overall specification difficult to port to another vendor interface.

The contribution of this paper is the addressing of the above issues related to the external memory interface of FPGA-based MPSoCs in a very innovative and efficient way. We propose a hierarchical memory system and an interface with a programmable controller to transfer data between external memories and on-chip memories using a DMA mechanism. The memory system and the interface allow automated programming, synchronization and scheduling of data dependent external memory accesses. The interface is generic and can be realized on FPGAs from different vendors. Our memory system and interface avoid the multi-master shared bus architecture and arbitration which results in better overall performance when a multiprocessor system is implemented on an FPGA.

1.1. Related Work

To the best of our knowledge there are not many references in the literature that address off-chip data communication related to multiprocessor systems implemented on FPGAs. The general approach is to connect a global off-chip memory to (on-chip) cache memories. This approach is inherited from the single processor architectures (widely used for more than 20 years) and implies lots of interprocessor synchronization challenges [3] such as cache coherency. Although the programming of such multiprocessor systems is relatively easy, the limited performance and cache coherency problems force us to devise alternative high-performance multiprocessor architectures with more efficient hierarchical memory and external memory interface.

The work presented in [4] proposes a memory system architecture with stream channels mapped onto an external memory and on-chip FIFOs, and schedulers for managing memory accesses. This is similar to our approach in the sense that we also propose a memory system that realizes FIFO communication (between processors

in an MPSoC). Their approach however supports a data stream execution model for designs with only a single computation task. In contrast, our hierarchical memory system is used to connect multiple processors where each processor executes its own computation task. In addition to that, our approach does not require application dependent knowledge to manage memory accesses (for communication between the processors) which simplifies the structure of our external memory interface.

Recent work on efficient FPGA multiprocessor system design for high throughput dataflow and stream oriented applications is presented in [5]. The targeted architecture consists of a network of processors connected through slow on-chip peripheral buses (OPBs), direct FSL links, and on-chip memories. The authors present an IPv4 packet forwarding application where the size and the performance of the system is limited by the amount of the on-chip memory and by arbitrating constraints to share the memory between different processors. The authors observed a significant drop in OPB performance if more than 2 processors share the same bus. The results presented in [5] fully support our statements that 1) for modern MPSoCs, using an external memory has to be considered and 2) defining an effective mechanism to share memory between different processors is an important aspect in designing high-performance MPSoCs.

1.2. Motivating Case

The work presented in this paper is an important extension of our methodology for automated multiprocessor system design, programming and implementation on FPGAs presented in [6]. The methodology has been implemented in a tool called ESPAM (Embedded System-level Platform synthesis and Application Mapping). We have experimented with ESPAM by implementing different MPSoCs on the Xilinx FPGA technology executing a Motion JPEG encoder application. Based on the obtained results presented in detail in [6] we have concluded that the main limitation on the size of a multiprocessor system that can be built on a single FPGA is the amount of the on-chip memory. For example, a system containing 2 $PowerPC$ and 2 $MicroBlaze$ processors utilizes 100% of the memory resources (BRAMs) and utilizes only 35% of the slices of a VirtexII-Pro 20 device [6]. Although, in every new Xilinx FPGA device the amount of the on-chip memory increases, still it remains a limiting factor. To overcome this, the usage of external (off-chip) memories has to be considered. However, using the FPGA on-chip memories instead of external memories is crucial for our high-performance multiprocessor systems because the external memories are slower than the on-chip BRAMs. In addition to that, the only interface provided by Xilinx to connect the $MicroBlaze$ and $PowerPC$ processors to an external memory is the slow OPB bus which reduces the performance even more if external memory is used. Therefore, we need to devise a different approach to use external memories in our platforms without sacrificing the overall performance.

In this paper we propose a hierarchical memory system that utilizes on-chip and off-chip memories for data communication between the processors in a multiprocessor platform. In our approach a single external memory is shared between different processors without limiting the overall performance. The hierarchical memory system uses custom buses and controllers for accessing the memories in a way that allows parallel operation of the processors without the need of arbitration of the memory accesses. We do not consider the usage of the shared external memory as a program and/or local data memory of processors because in such case arbitration of the accesses to the external memory can not be avoided. Such arbitration is crucial for the overall performance because if the processors compete between each other every time they fetch an instruction or (local) data, high performance can not be achieved. Local cache memories can be used in order to increase the performance of these systems. However, if cache memories are used for inter-processor data communication, cache coherence mechanisms has to be considered [3]. This introduces additional overhead limiting the overall performance.

2. MULTIPROCESSOR PLATFORMS

In this section we give a brief description of our approach, described in [6], for building multiprocessor platforms. The multiprocessor platforms we consider are constructed by connecting *Processing*, *Memory*, and *Communication* components using *Communication controllers (CC)*. *Memory* components are used to specify the processors' local program and data memories and to specify data communication memories (CM) used to transfer data between the processors. The *Communication* components determine the communication network topology of a multiprocessor platform, e.g., a point-to-point network, a crossbar switch, or a shared bus. The *Communication controller* implements an interface between processing, memory, and communication components. We have developed a general approach to connect and synchronize programmable processors of arbitrary types via a communication component.

Our approach is explained below using the example of a multiprocessor platform depicted in the right part of Figure 1 – see the non-shaded blocks. It contains several processors connected to a communication component (in this example – a crossbar switch CB) using communication memories (CMx') and communication controllers (CCx'). The processors transfer data between each other through the CMx' memories. A communication controller connects a communication memory to the data bus of the processor it belongs to and to a communication component. Each CC implements the processor's local bus-based access protocol to the CM for write operations and the access to the communication component (CB) for read operations. In our approach each processor writes only to its local communication memory and uses the communication component only to read data from all other communication memories. Thus, memory contention is avoided. Each CM is organized as one or more FIFO buffers. We have chosen such organization because, then, the inter-processor synchronization in the platform can be implemented in a very simple and efficient way by blocking read/write operations on empty/full FIFO buffers located in the communication memory.

The described communication mechanism and the usage of on-chip memories for communication results in high performance of our multiprocessor systems. However, for some applications the memory requirements may exceed the available (on-chip) memory resources of the system. Therefore, we devised a mechanism and extended our platforms to support external memory for inter-processor data communication. The remaining part of the paper describes this mechanism and our interface for an external memory.

3. HIERARCHICAL MEMORY SYSTEM

In this section we present our approach to use off-chip and on-chip memories, combined in a hierarchical memory system, for data communication between multiple processors in our platforms. An example of such platform is depicted in Figure 1. The hierarchical memory system is depicted in the left part of the figure – see the shaded blocks. For the sake of clarity the example contains only two proces-

sors. However, the number of processors is not limited to two. Each processor has its local Program/Data memory. Processors commu-

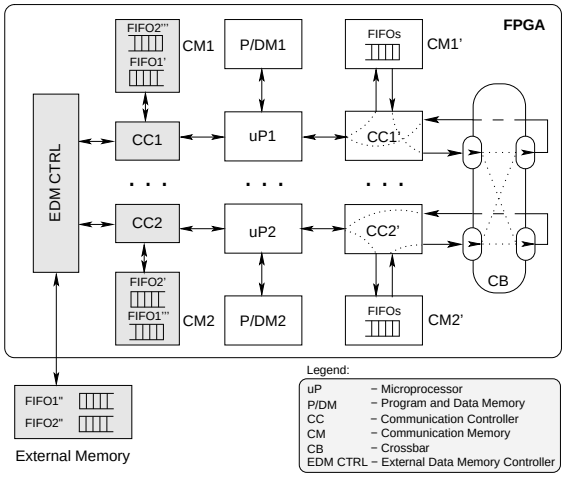


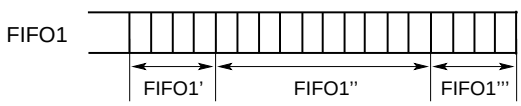
Fig. 1. Example of a Multiprocessor Platform.

nicate data only via FIFO buffers. Small size FIFOs are mapped onto on-chip memory $CM1'$ and $CM2'$ connected to a crossbar (CB) or other communication component. Large size FIFOs are mapped onto the hierarchical memory system that includes on-chip SRAMs, i.e., communication memories $CM1$ and $CM2$ and an external (off-chip) memory. Mapping of large FIFOs is explained in the next section. External Data Memory Controller (EDM CTRL) manages the access to the external memory. Also, EDM CTRL is responsible for synchronizing and moving data between the SRAMs and the external memory using a Direct Memory Access (DMA) mechanism.

Each processor has its own CM as a part of the hierarchical memory system (see the left part of Figure 1). The CMs are connected to communication controllers ($CC1$ and $CC2$). A CC allows multiple FIFOs to be mapped in a single CM and the CM to be accessed by the corresponding processor and EDM CTRL at the same time. The External Data Memory Controller and the CCs connect the on-chip CMs and the external memory in a particular way described in detail later in the paper. EDM CTRL acts as a single master (with CCs as slaves) that initiates data transfers between the CMs and the external memory.

3.1. Mapping FIFOs onto Hierarchical Memory System

As explained above, the hierarchical memory system consists of low latency on-chip communication memories (CMs) and a large off-chip memory. This memory system is used for mapping of large FIFO buffers as follows. For the sake of clarity assume that processors uP1 and uP2 (Figure 1) have to communicate data via large fifo buffers FIFO1 and FIFO2. Processor uP1 writes data to FIFO1 and reads data from FIFO2. Similarly, processor uP2 writes to FIFO2 and reads from FIFO1. In our approach each large fifo is split in three parts. For FIFO1, these are FIFO1', FIFO1", and FIFO1"' as shown in Figure 2 (FIFO2 is split in the same way). Each CM



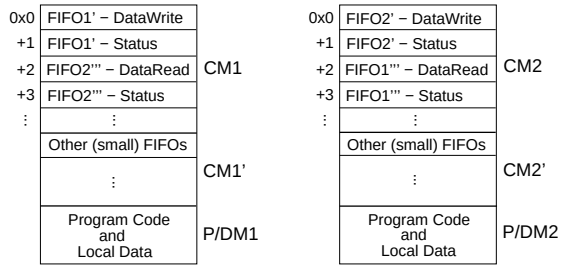
size(FIFO1) = size(FIFO1') + size(FIFO1") + size(FIFO1"')
size(FIFO1') = size(FIFO1"') << size(FIFO1")

Fig. 2. The FIFO structure.

is organized as several logical FIFO buffers. The off-chip memory is organized as logical FIFOs as well. FIFO1', FIFO2', FIFO1"', and FIFO2"' are small fifos. Therefore, they are physically mapped onto the on-chip CMs. FIFO1" and FIFO2" are large fifos and they are physically mapped onto the off-chip memory as shown in Figure 1. When processors uP1 and uP2 communicate data over the buffers (FIFO1 and FIFO2), they write/read only to/from the fifos FIFO1', FIFO2', FIFO1"', FIFO2"' located in the on-chip CMs. The processors synchronize by blocking write/read on these fifos. The External Data Memory Controller (EDM CTRL) is responsible for the physical movement of data between the on-chip CMs and the off-chip memory. EDM CTLR moves data from FIFO1' to FIFO1" to FIFO1"' , i.e., EDM CTRL reads data from FIFO1' and writes it to FIFO1" or EDM CTRL reads data from FIFO" and writes it to FIFO1"'. Similarly, it moves data from FIFO2' to FIFO2" to FIFO2"'.

3.2. Memory Maps

The multi-FIFO organization of the CMs is realized by the CCs. Each FIFO is seen by a processor as two memory locations in its memory address space. One location is used to read the status (full or empty) of a FIFO. The other location is used for reading/writing data from/to a FIFO. As explained above a FIFO is split in three parts (FIFO', FIFO", and FIFO"'). The memory maps in Figure 3 show that a processor can access only FIFO' and FIFO"' located in a CM. FIFO" located in the external memory cannot be accessed by a processor. It can be accessed only by the EDM controller.



a) Memory Map of uP1 **b)** Memory Map of uP2

Fig. 3. The Memory Maps of the Processors in Figure 1.

3.3. Inter-processor Synchronization and Data Transfer

The synchronization between the processors in the platform shown in Figure 1 is realized by blocking read/write mechanism on empty/full FIFOs as explained below. When a processor has to write data to a FIFO in the CM (e.g. processor uP1 to write to FIFO1'), the processor first checks if there is room in the corresponding FIFO. This is done by reading the FIFO status located at a memory address specified in the memory map of uP1. In Figure 3a the memory map of processor uP1 shows that the status of FIFO1' is located at address 1. If the FIFO is full, the processor blocks. Otherwise, it writes the data to the corresponding *DataWrite* address of the FIFO. For FIFO1' this is address 0 (see the same figure). When a processor has to read from a FIFO located in the CM memory (e.g. processor uP2 to read from FIFO1"'), it first checks if there is any data in the corresponding FIFO. This is done by reading address 3 (see the memory map of uP2, Figure 3b). The processor blocks if the FIFO is empty. Otherwise, it reads the data from memory address 2.

The data transfer between the local processors' FIFOs (e.g. FIFO1' and FIFO1"' in Figure 1) is realized by the External Data Memory Controller through the large part of the corresponding FIFO located in the external memory (e.g. FIFO1") using a DMA mechanism.

The controller has a table containing the location of each part of each FIFO. For our example these are FIFO1', FIFO1", FIFO1"', FIFO2', FIFO2", and FIFO2"'. EDM CTRL acts as a single master transferring data first between the on-chip memories and the external memory (i.e. all FIFOs' to all FIFOs") and then between the external memory and the on-chip memories (i.e. all FIFOs" to all FIFOs"'). The controller uses a Round-Robin policy to serve the FIFOs. In the beginning, the controller checks if there is data to be transferred, i.e., from FIFO1' to FIFO1". If FIFO1' is empty, the controller moves to the next FIFO in the table, i.e., FIFO2'. If FIFO1' is not empty, the EDM controller checks the status of the corresponding FIFO in the external memory (FIFO1"). If it is not full, a transfer of a predefined amount of data is initiated. If during the transfer the controller blocks on empty or full FIFO, then the transfer is suspended and EDM CTRL moves to the next FIFO in the table. After the last FIFO' in the table is served, the controller proceeds with moving data, using the mechanism described above, from the external memory to the on-chip memories, i.e., from all FIFOs" to all FIFOs"' according to the EDM CTRL table.

4. IMPLEMENTATION

In this section we give some details about the implementation of the Communication Controllers (CC) and the External Data Memory Controller (EDM CTRL) as part of the hierarchical memory system we propose. We use the Xilinx FPGA technology but any other FPGA technology can be used for implementation because our hierarchical memory system approach and external memory interface are general enough. Notice that in the hierarchical memory system the on-chip (CM) memories are implemented with the dual-ports Block Select RAM modules (BRAMs) available in the Xilinx FPGAs. The CMs are the only technology dependent components in our approach.

4.1. Communication Controller

The communication controllers ($CC1$ and $CC2$ in Figure 1) realize the interfaces of the processors and EDM CTRL for accessing the CMs. The CCs also implement the multi-FIFO behavior of each CM. We use the *FIFOs Unit* component, which is a part of the communication controller presented in [6]. The number of the FIFOs and the size of each FIFO are (generic) parameters of this component. For details about the multi-FIFO implementation we refer to [6]. Recall that in our approach for on-chip communication (the right part of Figure 1) each processor writes only to its local CM and uses other CMs only for read operations. Therefore, writing to a FIFO is performed only by the local processor and reading from a FIFO is performed only from the communication component side (other processors). In contrast, in the proposed hierarchical memory system (the left part of Figure 1) both a processor and the EDM controller may access a CM to read from and write to FIFOs. A contention occurs when both, a processor and the EDM CTRL, try to read from (or write to) a FIFO in a CM at the same time. Therefore, arbitration of a CM access is required. This means that while a processor reads/writes from/to its CM, EDM CTRL can not (and the other way round). Because of the temporal blocking due to arbitration, the overall performance of the system is reduced. To keep the high performance of our platforms we propose a CC that uses two memory banks per CM (see the right part of Figure 4). One bank is used for FIFOs written by a processor and the other for FIFOs written by EDM CTRL. This guarantees no contention when accessing a CM and therefore arbitration is not needed.

4.2. External Data Memory Controller

The structure of the external data memory controller is shown in Figure 4. To implement the multi-FIFO organization of the external memory we use the same component as in the communication controllers (CCs). The external memory of our prototyping board is a Micron Technology ZBT (Zero Bus Turnaround) Memory. The purpose of the ZBT Memory Port (the bottom of the EDM CTRL) is to couple the physical interface of the ZBT memory with the internal logic. The main control unit of the EDM CTRL is the Round-Robin Sequencer. It continuously serves all the ports connected to the CCs and controls the data transfers as described in Section 3.3. The sequencer first determines the action to be performed on all ports (signal $R/W\_Act.$). It can be either read from a (CC) port and write to the external memory, i.e., moving data from FIFOx' to FIFOx", or read from the external memory and write to a CC port, i.e., moving data from FIFOx" to FIFOx"'. The type of action is changed every time all the CC ports have been served.

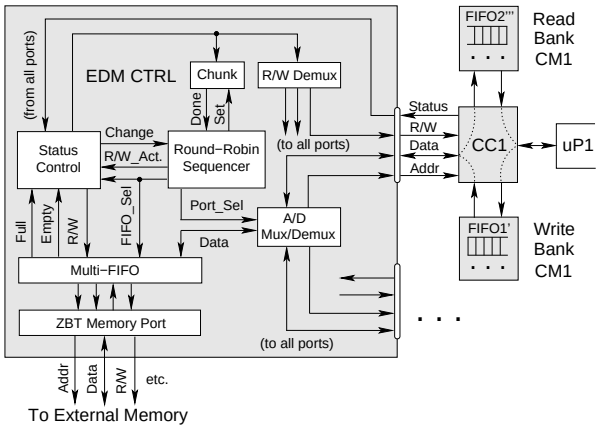


Fig. 4. The External Data Memory Controller.

Below we explain how a single port is served. The targeted FIFOs are specified by $Port\_Sel$ and $Fifo\_Sel$ signals. This is done according to a mapping table that determines the addresses of each FIFO', FIFO", and FIFO"'. The $Port\_Sel$ is translated in the $A/D\ Mux/Demux$ module to a physical address selecting a proper CC port and a FIFO (FIFOx' or FIFOx"') within the corresponding CM. Then, the status of the FIFO is read by the $Status\ Control$ module via the $Status$ signal. If a FIFO is accessed for read operation, then the status indicates *empty/not empty*. If the FIFO is accessed for write operation, then the status indicates *full/not full*. The read and write FIFOs are implemented in two physically separated memory banks (see $Read$ and $Write$ memory banks of $CM1$ in the right part of Figure 4). The communication controller ($CC1$ for the example in Figure 4) uses the most significant bit of the address ($Addr$) to select the proper memory bank and to generate the proper status of the corresponding FIFO. The status of the FIFO in a CM and the status of the corresponding FIFO" in the external memory is checked in the $Status\ Control$ module. If the FIFOs are neither empty nor full, then a transfer of a predefined amount (chunk) of data is initiated as the $Status\ Control$ module generates read/write strobe signals to the CC port and the external memory. These signals are coordinated in a way that takes into account the difference in the timing characteristics of both BRAM and ZBT memories. If the transfer is not possible, or the transfer blocks on a full or empty FIFO, the $Status\ Control$ module generates $Change$ signal that forces the sequencer to move to the next port (or the next FIFO of the same CC port) according to the mapping table. After the predefined amount of data has been transferred, the

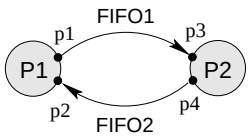
$Chunk$ module generates $Done$ signal to the sequencer and the latter moves to the next FIFO in its table. For each data transfer the chunk size is set by the Round-Robin sequencer (signal Set) to be equal to the corresponding FIFO' (or FIFO"') size.

5. AUTOMATED PROGRAMMING

Our methodology for multiprocessor system design, implemented in a tool called ESPAM and presented in [6][7], allows systematic and automated synthesis, programming, and implementation of MPSoC platforms. Automated programming of an MPSoC means that our tool automatically generates program code for each processor in the system, generates the memory map of the system, and generates code that implements the synchronization and communication between the processors. In this section we describe how we extended ESPAM to support automated programming of MPSoC platforms when FIFO channels are mapped onto the proposed hierarchical memory system.

5.1. Programming model and code for processors

In our approach for design, implementation, and programming multiprocessor systems, we use the Kahn Process Network [8] MoC as a programming model. In general, a Kahn Process Network (KPN) is a network of concurrent autonomous processes that communicate data in a point-to-point fashion over unbounded FIFO channels, using a blocking-read on an empty FIFO as synchronization mechanism. Because in the implementations the FIFO channels are finite in size, we use a blocking-write on full FIFO as well. A simple example of a KPN is shown in the left part of Figure 5. Two processes $P1$ and $P2$ are connected through FIFO1 and FIFO2. In the bottom-left part of the same figure we show a simple example of code for process $P1$ generated by ESPAM to be executed by a processor. The process reads data from its input channel via port *p2* (line 3). If data is not available, then the process blocks on reading until data arrives. Then it performs a computation on the data (line 4), and writes the result to its output channel via port *p1* (line 5). Lines 3 to 5 are repeated several times. The blocking read/write synchro-




```
1  void main() { // process P1
2     for ( int k=1; k<=N; k++ )      {
3        read( p2, in_0, sizeof(myType) );
4        compute( in_0, out_0 );
5        write( p1, out_0, sizeof(myType) );
6     }
7  }
```

```
8  void read( byte *port, void *data, int length ) {
9     int *isEmpty = port + 1;
10    for ( int i=o; i<length; i++ ){
11       // reading is blocked if a FIFO is empty
12       while( *isEmpty ){ }
13       (byte* data)[i] = *port; // read data from a FIFO
14    }
15 }
16
17 void write( byte *port, void *data, int length ) {
18    int *isFull = port + 1;
19    for ( int i=o; i<length; i++ ){
20       // writing is blocked if a FIFO is full
21       while( *isFull ){ }
22       *port = (byte* data)[i]; // write data to a FIFO
23    }
24 }
```

Fig. 5. A KPN example and code generated by ESPAM.

nization primitives are depicted in the right part of Figure 5. They are automatically generated and inserted in the program code by ESPAM in the places where a processor has to read/write data from/to a FIFO channel. Both primitives have 3 parameters: $port$, $data$, and $length$. Parameter $port$ is the address of the memory location through which a processor can access a given FIFO. Parameter $data$ is a pointer to a local variable and $length$ specifies the amount of data (in bytes) to be transferred. Recall that a FIFO is seen by a processor as two memory locations in its memory address space. A processor uses the first location to read/write data from/to the FIFO and the second location to read its (empty/full) status. The primitives implement the blocking synchronization mechanism in the following way. First, the status of a channel that has to be read/written is checked. A channel status is accessed using the locations defined in lines 9 and 18. The blocking is implemented by while loops with empty bodies in lines 12 and 21. A loop iterates (does nothing) while a channel is full or empty. Then, in lines 13 and 22 the actual data transfer is performed.

5.2. Memory map and EDM controller mapping table

Each FIFO in our MPSoCs has separate read and write ports. A processor accesses a FIFO using the *read/write* synchronization primitives described in Section 5.1 where the parameter $port$ specifies the address of the read/write port of the FIFO. The FIFOs are implemented in the communication memories, therefore the addresses of the FIFO ports are located in the processors' address space where the communication memory segment is defined (see Figure 3). The memory map of a MPSoC generated by ESPAM contains the values defining the read and the write addresses of each FIFO in the system. In our example there are two FIFOs (FIFO1 and FIFO2). Assume that process $P1$ is executed by processor $uP1$ and $P2$ by $uP2$ of the platform in Figure 1. Because the FIFOs are mapped onto the hierarchical memory system, the memory map of the MPSoC consists of the read and write addresses of their corresponding FIFOx' and FIFOx"' parts. The memory map as generated by ESPAM is shown in Figure 6a. Notice that FIFO1', FIFO2' have equal write addresses. This is not a problem because writing to these FIFOs is done by different processors and these FIFOs are located in the local CMs of these different processors, i.e., these addresses are local processor write addresses. The same applies for the read addresses of FIFO1"' and FIFO2"'.

```
#ifndef _MEMORYMAP_H_
#define _MEMORYMAP_H_

#define p1 0x0 // write address FIFO1';
#define p2 0x2 // read address FIFO2"';
#define p4 0x0 // write address FIFO2';
#define p3 0x2 // read address FIFO1"';

#endif
```

a) MPSoC memory map

	FIFO"	FIFO'			FIFO"'		
FIFO1	1	1	1	0	2	1	1
FIFO2	2	2	1	0	1	1	1
		#M	#F	A	#M	#F	A

b) EDM CTRL mapping table

Fig. 6. FIFOs Memory Map and EDM CTRL Mapping Table.

The mapping table of the EDM controller, described above, is generated automatically by ESPAM as well. It is shown in Figure 6b. It gives the relation between the FIFOs in the external memory (FIFOx") and the corresponding FIFOx' and FIFOx"' parts located in the on-chip CMs. In the mapping table these FIFOs are associated with three numbers: a number of a CM (#M), a number of a FIFO in this CM (#F), and a number specifying the access of this FIFO by EDM CTRL, i.e., A = '0' for read and A = '1' for write. Consider for example FIFO1. It is the first FIFO in the external memory, i.e., FIFO1" = 1. FIFO1' is read by EDM CTRL and written by $uP1$. Also, FIFO1' is the first FIFO in the write bank of $CM1$ (Figure 4). Therefore, the numbers in the mapping table related to FIFO1' are 1 1 0. Similarly, FIFO"' is accessed by EDM CTRL for write operations. Also, FIFO1"' is the first FIFO in the read bank of $CM2$. Therefore, the numbers in the table related to FIFO1"' are 2 1 1.

6. EXPERIMENTS AND RESULTS

In this section we present an experiment and the results obtained by implementing and executing a Motion JPEG (M-JPEG) encoder application onto two alternative MPSoCs using our ESPAM tool. As processing components we used $MicroBlaze$ processors. Both MPSoCs have been tested on a prototyping board with a Xilinx VirtexII-6000 FPGA. The purpose of this experiment is to compare the achieved results, in terms of resource utilization and execution

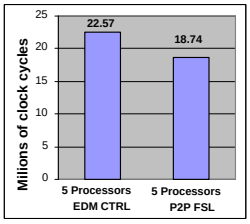
performance, between a multiprocessor system using the proposed hierarchical memory system and a system using only on-chip memory resources for data communication. For the latter we chose an MPSoC with a point-to-point (P2P) communication topology using dedicated HW FIFO components (the Xilinx' FSLs) because the point-to-point communication guarantees the highest (communication) performance [6].

Synthesis Results. The table in Figure 7a shows the overall resource utilization of the two MPSoCs we consider in our experiment. The table shows that a 5-processor P2P system utilizes 96% of the available on-chip memory (BRAMs). The other FPGA resources are grouped into slices that contain 4-Input Look-Up tables and Flip-Flops. The first three rows in the table (second column, 5 Proc P2P) show that this system utilizes only 14% of the slices, 5% FFs, and 8% LUTs. Although, there are plenty of slices to add more $MicroBlaze$ processors to the system this is not possible due to the lack of on-chip BRAM memory. In contrast, comparing the above numbers with the resource utilization of the MPSoC using our hierarchical memory system with 5 processors (see columns 5 Proc EDM), show savings of 31% of the on-chip BRAM components. At the same time, this MPSoC utilizes only 5% more slices for implementation of the hierarchical memory system. The difference in the amount of resources utilized by both systems is shown on the right most column of the table. These numbers clearly indicate that our approach to map communication buffers onto an external memory using the proposed hierarchical memory system is very efficient in terms of resource utilization, thereby allowing larger MPSoCs to be built.

Xilinx VirtexII-6000 FPGA

#Flip-Flops	3678	5%	4941	7%	+2%
#4-Input LUT	5822	8%	9146	12%	+4%
#Slices	4763	14%	6443	19%	+5%
#BRAMs	139	96%	94	65%	-31%
	5 Proc P2P		5 Proc EDM		

a) Resource Utilization



b) Performance Results

Fig. 7. Resource Utilization and Performance Results.

Performance Results. The numbers presented in this subsection are collected by running both M-JPEG multiprocessor system implementations on our FPGA board. For each MPSoC we measured the exact number of clock cycles needed to process one image frame of size 128 by 128 pixels. These numbers, depicted in Figure 7b, are taken from simple hardwired timers automatically integrated by ESPAM in each system. The left bar shows the performance of the MPSoC using the proposed hierarchical memory system for data communication. The M-JPEG application contains 13 FIFO channels, in this case, all mapped onto a single off-chip memory which is the worst case scenario possible. The numbers in Figure 7b show 16.9% drop in performance compared to the 5-processor P2P system (22.57M vs 18.74M clock cycles) which is an acceptable trade-off — recall that we save 31% of the BRAMs. We consider the performance of the hierarchical memory system to be very efficient because 1) the off-chip (ZBT) memory of our prototyping board is more than 2 times slower than the on-chip BRAMs, i.e., if faster off-chip memory is used, the performance will be better; 2) a single off-chip memory is shared between 5 processors, and at the same time the achieved performance is close to the performance of 5-processor system using distributed memories. Recall that our approach targets splitting of the communication, i.e., large channels are mapped onto (slower) off-chip memory and small channels onto (faster) on-chip memories. This separation leads to better performance compared to the worst case scenario we show in our experiment. Notice however that because of the memory requirements of some applications this worst case scenario, i.e., using only an off-chip memory for data communication, might be the only possible approach to implement these applications using MPSoCs.

7. CONCLUSION

In this paper we presented our general approach (implemented in the ESPAM tool) to use an external (off-chip) memory for inter-processor data communication in multiprocessor systems on FPGAs. While the state-of-the-art development tools support off-chip memory access based on multi-master shared bus architectures (with or without DMA), we propose a hierarchical memory system that uses a programmable controller to transfer data between the off-chip and on-chip memories using DMA mechanism. Our approach guarantees no contention when a local memory is accessed by a processor and the DMA controller at the same time. Therefore, no arbitration is needed and no communication overhead is introduced which leads to better overall performance.

The results presented in this paper show that our approach of connecting processors through the proposed hierarchical memory system is efficient in terms of resource utilization and performance. For an M-JPEG encoder application mapped onto 5 $MicroBlaze$ MPSoC with our hierarchical memory system, the implementation of this memory system adds only 5% to the utilized FPGA resources and saves 31% of the on-chip BRAMs compared to 5 $MicroBlaze$ MPSoC which does not use external (off-chip) memory. The on-chip memory saving allows larger MPSoCs to be built.

8. REFERENCES

[1] "Xilinx, Inc. XPS and EDK, version 8.1i edition.," www.xilinx.com/ise/embedded_design_prod/platform_studio.htm.

[2] "Altera, Inc. Quartus II Handbook Volume 4: SOPC Builder, Dec 2005.," www.altera.com/literature/quartus2/lit-qts-sopc.jsp.

[3] Paul Stravers and Jan Hoogerbrugger, "Homogeneous Multiprocessing and the Future of Silicon Design Paradigms," in *Proc. Int. Symposium on VLSI Technology, Systems, and Applications (VLSI-TAS'2001)*, 2001, pp. 184–187.

[4] Joonseok Park and Pedro Dinis, "An External Memory Interface for FPGA-Based Computing Engines," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, California, USA, 2001.

[5] K. Keutzer et al., "An FPGA-based Soft Multiprocessor System for IPv4 Packet Forwarding," in *Proc. FPL*, Finland, 2005.

[6] Hristo Nikolov, Todor Stefanov, and Ed Deprettere, "Efficient Automated Synthesis, Programming, and Implementation of Multi-processor Platforms on FPGA Chips," in *Proc. 16th Int. Conference on Field Programmable Logic and Applications (FPL'06)*, Madrid, Spain, Aug. 28-30 2006, pp. 323–328.

[7] Hristo Nikolov, Todor Stefanov, and Ed Deprettere, "Multi-processor System Design with ESPAM," in *Proc. 4th IEEE/ACM/IFIP Int. Conf. on HW/SW Codesign and System Synthesis (CODES-ISSS'06)*, Seoul, Korea, Oct. 22-25 2006, pp. 211–216.

[8] Gilles Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proc. IFIP Congress*. 1974, North-Holland Publishing Co.