Building a Flexible and Scalable DRAM Interface for Networking Applications on FPGAs

Jike Chong, *Student Member, IEEE,* Chidamber Kulkarni, *Member, IEEE*, and Gordon Brebner

Abstract—A fundamental challenge to successful deployment of DRAMs is the availability of a flexible and scalable DRAM interface. This is exacerbated by the application specific nature of the logic-side DRAM interface. This paper presents a study that attempts to overcome this challenge for networking application domain. We quantify the various challenges and present techniques that were implemented to build a flexible and scalable interface to an existing multi-port memory controller for DDR DRAM using a FPGA. We demonstrate the deployment of this new interface in two example applications. We present two novel techniques that enable us to reduce the latency of DRAM related memory accesses and improve throughput. We believe our techniques enable harnessing maximum throughput from existing memory controllers with least possible latency.

Index Terms—Memory architecture, Memory management, Routing, Parallel processing

I. INTRODUCTION

THE search for application-specific solutions with ever decreasing time-to-market is pushing system designers away from the risky time-consuming ASIC design process towards programmable platforms, such as FPGAs. In addition, network line cards require external memories, such as DRAM, for many reasons (e.g. storing look up tables, shunting packet payloads, storing packet statistics, etc). In practice a complete network line card could require a dozen or more such external memory chips. Thus, integrating a malleable memory interface into an FPGA-based networking application design flow is essential.

FPGAs are well suited for interfacing with external memory(-ies). The logic-side interfaces for such memory controllers are application specific and need to be adapted for different applications as well as for integration within an overall design. Due to their programmability, FPGAs are an attractive choice. However, integrating memory controllers for a given application scenario becomes tedious, error-prone, and time consuming due to the lack of a flexible and scalable interface. In fact, the memory interface becomes a design bottleneck if appropriate care is not taken in the overall design process. Building a scalable and extensible interface is non trivial because of two aspects: first, there is a vast variety of external memories available in the market today, and second, different application domains have different characteristics that impacts the type of optimizations one can apply.

SRAM and DRAM memories have synchronous interfaces for the control and data buses. Internally these memories have multiple memory banks and the input/output data bus is shared between different memory banks. DRAM architectures support varying degrees of control over concurrency internally. For example there are control instructions that allow activation of a row to perform consecutive read or write operation or allowing combined read or write operation with pre-charge operation, etc. An efficient memory controller matches the concurrency required by the application to that available in the DRAM architecture.

Application domains also have a significant impact on the logic-side interface to DRAM controllers. For example, in network processing, the interface needs to support signals for start, pause, or end of a packet. Another example of application specific characteristic is the elastic buffer requirement for network processing. Sizes of network packets on a network vary, whereas sizes of frames for a segment of video do not. In addition, the amount of concurrency one can exploit for network processing is much higher than e.g. video processing. Such concerns significantly impact the design and scaling of a logic-side interface for servicing varying number of (concurrent) state machines. One of the main motivations for our study is the need for domain specific considerations and optimizations of the logic-side interface. In this paper we focus on network processing as an example application domain.

The design productivity gap is widening with every new process generation. To close this gap, we need methods and tools that can transform higher-level concurrent semantics into efficient HDL implementations. This work contributes to building a flexible and scaleable DRAM memory interface. Such an interface can be easily incorporated in to

Jike Chong is a Ph.D. student at University of California, Berkeley, Berkeley, CA, USA (e-mail: Jike@eecs.berkeley.edu)

Chidamber Kulkarni is a researcher at at Xilinx Research Labs, Xilinx, Inc. San Jose, CA, USA (e-mail: Chidamber.Kulkarni@xilinx.com)

Gordon Brebner is a distinguished engineer at Xilinx Research Labs, Xilinx, Inc. San Jose, CA, USA (e-mail: Gordon.Brebner@xilinx.com)

different networking application scenarios specified using a domain specific language such as Click [1]. Specifically, it should a) allow an arbitrary number of threads of control to access DRAM,b) harness maximum throughput from the DRAM bus, and c) support different quality of service to different ports tunable to different application scenarios.

In the remainder of the paper, we will present related prior work, an extensible interface that we have implemented, and experimental results on an Internet Protocol routing application. In particular we provide a detailed analysis of various latencies in the pipeline and approaches to reduce them. We end the paper with the main conclusions and directions for future work.

II. Background and related work

Memory controllers have received significant attention in the research domain [2], [3], [4], [11]. The majority of the existing efforts have been targeted at maximization of DRAM throughput by re-ordering of memory accesses based on the target address within the DRAM. In particular most of the mainstream effort has been on optimizing the throughput for a single stream of control for servicing a single processor cache hierarchy [2], [3], [4], [11].

FPGA-based designs attain better performance by exploiting concurrency, and so have multiple threads of control interfacing to the memory controller. Thus, adapting the DRAM interface for varying number of threads of control is a critical requirement. For example, on the Xilinx website [5] alone, there are more than a dozen different DRAM controllers for different applications and DRAM architectures. This demonstrates the conundrum the system designer faces when choosing a particular memory controller for external memories.

Park and Diniz [6] present an approach to synthesize and estimate performance of a SDRAM memory controller for FPGA-based applications. An important aspect of their approach is the decoupling of the external memory controller from the application related data path design. This allows for easy composition based on a bus-like interconnection structure. However it does not efficiently utilize the specific characteristics of any particular application domain.

An investigation of input and output queuing schemes for DRAM based router designs for ASIC implementations has been done by McKeown et al [7]. We build upon their conclusions in terms of choosing a particular style of memory allocation scheme. Sherwood et al [10] present a wide word pipeline DRAM architecture suitable for layer 3 and below network processing applications. Hasan et al [11] present memory access re-ordering techniques for a fixed network processor architecture.

In this work we leverage an existing DDR SDRAM memory controller [8] called the Multi-Port Memory Controller (MPMC) available as a Xilinx reference design. It is important to note that the MPMC was designed for a networking application domain, thus we leverage some of the optimizations performed for a single channel controller for the networking application domain. Our goal is to build layers of abstraction suited for the networking application domain upon this existing (and efficient) single channel memory controller. In the following sections we will explain the different layers of abstraction that we have implemented on top of MPMC to build a flexible and scalable interface to DRAM memory.

III. Introduction to MPMC

The MPMC is a quad-port DDR SDRAM memory controller that can arbitrate four concurrent streams of control to access memory. The logic-side interface has four ports for 64-bit data. The memory side has one physical interface for 32-bit DDR SDRAM with a 125MHz system clock (250MHz data rate). A 32-bit bus running at 250MHz offers a maximum throughput of 8 Gbit/s. Figure 1 depicts the architecture of MPMC. The DDR interface sequentially transmits controls and data on the bus. With various control overhead, the maximum achievable throughput is somewhat lower, as discussed later.
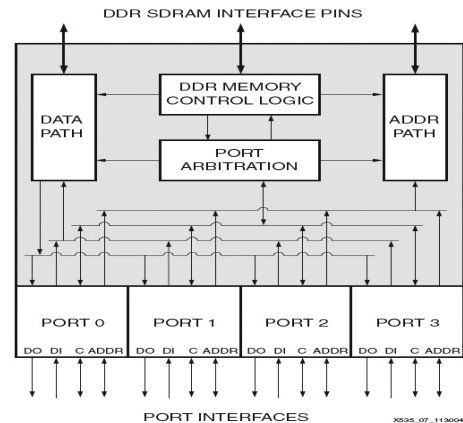

Fig. 1. MPMC and internal components

MPMC supports four memory access modes, ranging from a single (32-bit) word access to a 32-word burst access. The mode is selectable with each access to the memory. Inside the MPMC, the four memory ports are arbitrated onto the DRAM bus. Each memory port has a read data buffer and a write data buffer of 32 words, sized to store enough data for one burst access.

In each memory write, the data and controls are presented at an MPMC port with a request for service. When there is space in the write data buffer (burst buffer) for burst access, MPMC acknowledges the request, and transfers the data into the burst buffer. The data stays in

this buffer until the memory bus is available. Pre-coded control sequences, stored in on-chip block RAM, are used for the physical interface controls. When the memory bus is available, a control sequence is retrieved from command memory in MPMC and placed on the DRAM bus, then followed by the data.

In each memory read, the request is presented at an MPMC port. When the memory bus is available, the appropriate pre-coded control sequence is retrieved and sent to the DRAM. When the DRAM responds with the data, the data is transferred into the port's read burst buffer. Data in the burst buffer can be read out as soon as the first word is available from memory. The read operation at each port is partially pipelined between memory bus access and burst buffer access.

Each memory port can service either a read or a write access at a time, as the address control is shared between read and write. The four memory ports can operate independently, i.e. one can be servicing a read or write, while another is servicing a different read or write. Access to the memory bus is arbitrated between all ports in a round-robin fashion.

There is significant latency overhead in accessing the DRAM. Among the four memory access modes, single word access has the most overhead per data word accessed. The 32-word burst has the least overhead per data word accessed. Even for a 32-word burst write, various latencies add up to 42 cycles. Specifically, there are 19 cycles to load the burst buffer, 2 cycles for DRAM bank activation, 16 cycles for data transfer on the DRAM bus, and 5 cycles for DRAM pre-charge. Read and write accesses have different amounts of overhead. If there is an equal number of read and write accesses, as in a buffering scenario where the DRAM is used as a FIFO, sequentially accessing DRAM achieves a bus throughput of 3.51 Gbit/s, or 44% of peak throughput.

Taking advantage of the independent nature of the MPMC memory ports, we can allow memory accesses to DRAM to be interleaved, such that one port could be loading its burst buffer, while another is utilizing the DRAM bus. This interleaving technique can recover DRAM bus throughput to 5.95 Gbit/s, or 75% of peak throughput. Table 1 lists the different latencies observed for the MPMC.

In the following section we discuss the implementation of an abstraction layer on top of MPMC to make it extensible to arbitrary number of ports.

IV. EXTENSIBLE INTERFACE

An extensible interface to the DRAM memory controller is required to support concurrent FSMs in the logic fabric. An extensible DRAM interface should a) allow an arbitrary

TABLE 1
DRAM BUS THROUGHPUT AND LATENCY

(interleaving /sequential)	Read (Cycles)	Write (Cycles)	Total (Cycles)	Throughput (Gbit/sec)
Word (32 bits)	7 / 14	8 / 13	15 / 27	0.53 / 0.30
4-Word (128 bits)	8 / 15	9 / 15	17 / 30	1.88 / 1.07
8-Word (256 bits)	8 / 17	11 / 19	19 / 36	3.37 / 1.78
32-Word (1024 bits)	20 / 29	23 / 44	43 / 73	5.95 / 3.51

number of threads of control to access DRAM, and b) harness maximum throughput from the DRAM bus.

To gain a quantitative understanding of the trade-offs for such a memory interface, we leverage the validated MPMC as a building block for a memory interface. A fundamental feature of our approach is to build layers of abstraction in supporting a memory controller for external memory. For example, the lowest layer supports a single channel DDR DRAM controller that controls the physical interface to a DDR DRAM memory. On top of this is the MPMC, configured in our study to support two read-only and two write-only ports. The significance of this configuration is explained later in this section. With this configuration, the interface is able to harness maximum throughput from the single channel memory controller. However, to make it easy for integration within the application domain, we introduce a layer of abstraction that supports an arbitrary number of input and output ports with an interface suitable for networking applications.

One alternative approach is to build a single layer of logic (single memory controller) that implements all three layers of abstraction described above. Such an approach might potentially have lower latency but will be difficult to maintain, reuse, and adapt to different application domains.

Another alternative approach is to combine the top two layers of abstraction, and have the application developer design to the single channel controller. In such an approach, the application developer losses a standard memory interface, and must then explicitly manage burst operations, and explicitly exploit concurrencies between burst buffer load and physical interface access for multiple independent memory accesses. We maintain that the functions in MPMC best managed by memory experts, and orthogonalized to application specific optimizations.

To support arbitrary numbers of ports in a memory interface, a simple approach is to statically map each thread of control to one of the four MPMC memory ports, such that each MPMC port will service one fourth of the threads of control. Such a topology requires less arbitration logic per port, but has a severe throughput penalty: DRAM accesses that are mapped onto one port are serialized. If most of the active threads are mapped to one of the ports,

the serialized accesses cannot be interleaved by the MPMC, and the maximum achievable throughput will drop dramatically.

An extensible approach is to use an arbiter to dynamically arbitrate input requests onto the MPMC ports. Facing the logic, there could be an arbitrary number of control threads connected to the arbiter. Facing the MPMC, the write requests are mapped to two of the MPMC ports, and the read requests are mapped to the other two MPMC ports.

The MPMC burst buffer access time is shorter than DRAM access time for 32-word bursts. Two-stage interleaving is enough to keep the DRAM bus maximally utilized. With two ports for each type of accesses, one port could be accessing the burst buffer, while another utilizes the DRAM bus. This pipelining of accesses guarantee interleaved memory access whenever two or more requests are present.

The arbitration of FSM threads of control onto the MPMC ports can be adapted for specific application scenarios. Typical arbitration schemes found in networking applications such as deficit round robin and weighted fair queuing can be employed to optimize for particular network traffic patterns. This flexibility in the arbitration scheme only impacts the extensible layer component, not the MPMC, nor the single channel memory controller. Figure 2 illustrates the extensible approach we implemented, contrasting it with the simple approach. When utilizing future faster DRAM buses, burst buffer accesses may take longer than the transmission of a burst of data. For such external memories, more than two stages of pipelining may be required to maximize utilization of the DRAM bus. However, this will not affect the dedicated single channel controller for the particular DRAM memory.

V. CASE STUDY: INTERNET PROTOCOL ROUTING

Applying the extensible DRAM interface to a networking application, we chose the payload shunting application in a gigabit Internet Protocol (IP) router to stress test throughput of the memory sub-system.

In our case study we investigate the split header and payload architecture. This approach enables higher utilization of header processing logic. Figure 3 illustrates this architecture for a two port IP router application, which can be generalized to N ports.

In a router, packets from any input port must be able to go to any output port. The obvious approach is to implement a fully connected crossbar. However, this approach is not scalable. For example, in a sixteen port router, there needs to be 16x16=256 buses of N bits wide to implement a fully connected crossbar. Such a routing resource can significantly constrain implementation
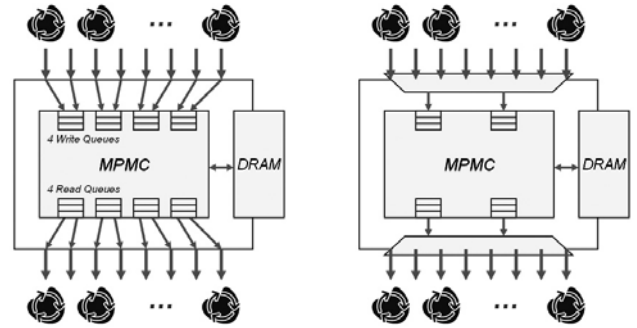

Fig. 2. Simple approach (left) vs. extensible approach (right)

alternatives; even in today's rich FPGA interconnect fabric. In addition, meeting timing can become a challenge due to large routing overhead.

Using a memory as a bus-based crossbar mitigates this routing problem. When a packet is written into a memory location from one port, it can be read out to any other ports. An additional port for the router only requires additional routing to and from the memory port. This linear scaling of the routing resources is much more manageable than the quadratic scaling for a cross bar. With multi-gigabits per second throughput on the memory bus, it is possible to multiplex several concurrent streams of control on a DRAM bus for sequential access to the DRAM memory.
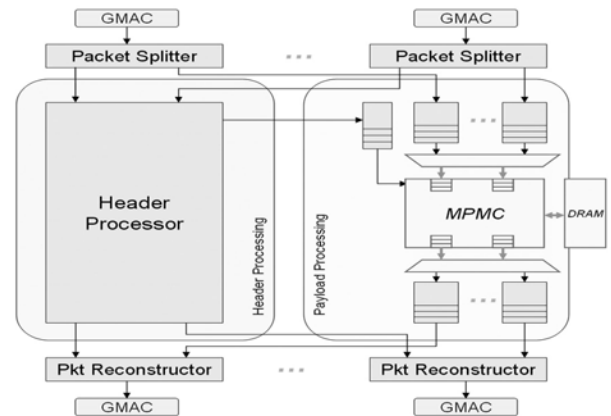

Fig. 3. Split header/payload processing

A. Payload shunting using the extensible memory interface

In the IP router application, external memory is used to buffer in-flight packet payloads. The application constrains packets from the same input port to be processed in order of arrival. This implies FIFO-like behavior at the memory interface. The FIFO queue abstraction has input data, output data, and read/write controls. Memory addresses have to be generated and managed by the interface to keep track of locations of the payload packets. During each routing operation, a piece of memory is allocated for a packet, and the payload is written into the memory. The

location and packet length are passed as descriptors to the destination port to be queued and retrieved. The three main memory allocation schemes that we considered are: global queuing, input queuing, and output queuing. Output queuing was selected as the method of implementation. We discuss the memory allocation scheme further in the following section.

1) Global FIFO Queuing: In global queuing, memory addresses are allocated following the sequential ordering from the header processor. Each payload is allocated a consecutive memory block, and is followed immediately by the next payload in the sequential ordering.

This memory allocation strategy is easy to compute. The next payload starting address is the current payload starting address plus current payload size. There is no gap between payloads in memory, which allows maximum utilization of available memory space. However, it suffers from the head of line (HOL) problem. If the head of the FIFO is blocked by resource contention on one output port, no other elements in the queue can proceed before the blockage is resolved. Figure 4 illustrates the HOL blocking case where the second entry in the FIFO (labeled 1) is destined for output port 1 but is blocked since the head of FIFO is destined for output port 2 which is busy transmitting another packet.

If we allow out-of-order processing of the FIFO queue, memory could be freed out-of-order, leading to memory fragmentation, which may imply complex garbage collection routines requiring inappropriately large FPGA resource usage.

2) Input FIFO Queuing: In input queuing, memory is segmented into equal partitions dedicated to each input port. Memory addresses are allocated consecutively following the sequential ordering of each input. The payload starting address is computed per input port.

Memory is less efficiently utilized as packets may be dropped when there is still free memory space. Packets at an input may be dropped when its own input queue is full while other input queues may still have space. The HOL blocking problem still exists, but is less serious in the average case. There are many heads of queue to route from, so the HOL problem affects each input separately. Figure 4 illustrates the HOL blocking case for input queuing. The left-most input queue has an entry (packet) destined for output port 0. But it is blocked by the head of FIFO entry that is destined for output port 2 which is busy transmitting.

Attempting to free memory out-of-order would be more complex because there is more than one queue to maintain.

3) Output FIFO Queuing: In output queuing, memory is segmented into equal partitions dedicated for each output port. The destination port is known before the payload is written to memory. In other words, packet payloads are stored into memory in the order they become free. This

completely resolves the HOL blocking problem and no memory fragmentation will occur. Figure 4 illustrates the output queuing scheme.

Memory utilization efficiency is the same as input FIFO queuing, and not as efficient as global FIFO queuing. Header IP lookup must precede memory allocation which, depending on implementation, may delay the write to the memory and increase the input queue size requirement.
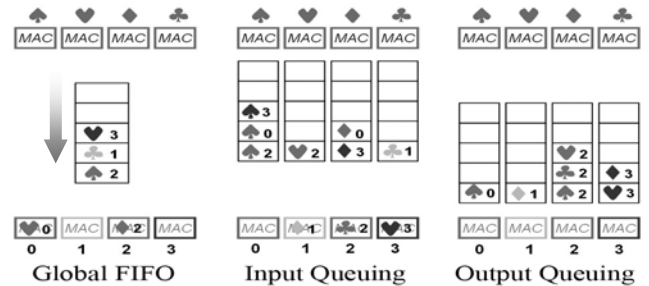

Fig. 4. Global, input and output queuing schemes

In traditional output queuing systems, the output queue must have r-speedup (r times as fast as line-rate), as multiple input ports can be writing to the same output queue concurrently. In the memory-based cross bar architecture, the requirement is naturally satisfied, as multiple input ports can interleave writes to distinct memory addresses in a single output queue partition.

Output queuing was selected here for its straightforward implementation of memory allocation and freeing schemes. It trades additional latency in allocating memory for ease of memory freeing.

B. Experiments

A two port router was implemented, with payloads being shunted to off-chip DRAM. It was also extended to a four port router version to illustrate the effect of scaling.

The implemented routers have the same top-level architecture. Packets come into the fabric through Gigabit Medium Access Control (GMAC) ports. Each GMAC is a port of the router that has separate gigabit transmit and receive channels. For the receive channel, each port has a packet splitter module that separates the packet header from the payload. The headers are directed to one shared header processing unit, and the payloads are shunted to the shared payload processing unit, with dedicated queues for each GMAC. The header processing unit does IP address lookup, and indicates to the payload processing unit to which destination port the payload should be routed. Based on the destination port, the payload processing unit computes an address in the memory partition of the destination output queue in DRAM. An FSM is then invoked to write the payload to this address. The descriptor containing this address is passed to the destination port descriptor queue. The output descriptor

queue is drained by an output port FSM which retrieves the payload from DRAM, through MPMC, into the output payload buffers. The packet reconstructor modules then unite the payloads with their respective headers, and the reconstructed packets are sent out by the GMACs on the transmit channel. The GMAC has a physical layer facing the FPGA I/Os and a logic side facing the packet splitters and reconstructors.

Header processing is not the emphasis of this study, so static IP address lookup sufficed in the current implementation. Since the payloads are written to memory in 32-word bursts, it takes 128 cycles to accumulate enough data for a burst. Header processing can use this time to implement these approaches as alternatives without affecting performance.[1] The header processing can easily be shared among all inputs by pipelining the lookups. Investigation of DRAM-based lookup remains an active research topic.

In payload handling, a packet is first buffered in the input FIFO queues implemented with on-chip Block RAM (BRAM). Once the memory destination address is known, data in the input FIFO queue is written to external memory one full burst at a time.

Each input port has an FSM producing control sequences for the MPMC port. The input arbiter arbitrates between all ready input FSMs. Access to the MPMC is granted by providing a communication channel from an input FSM to a MPMC port. By having an FSM for each input port, we are able to interleave burst controls from multiple input ports at the granularity of a single burst length.

C. Results for the extensible interface

The designs were targeted at the Xilinx Virtex-II Pro series FPGAs, and specifically implemented on a XC2VP30 device. The main timing constraint of 125MHz was met after placement and routing. The designs were verified in RTL simulation and the numbers measured and reported here are from RTL simulation using sample network traffic patterns of a mix of three packet sizes: small, medium, and large, containing one, five and fourteen 32-word burst length of payload respectively. To maximize throughput on the DRAM memory bus, the design exclusively uses 32-word bursts. Sub-burst length payload is not handled in this implementation, although multiple payloads could be concatenated into one burst, requiring additional control logic to maintain the payload descriptor.

Throughput was measured by recording the time spent on the memory bus over 100 burst accesses. Since the router is designed to harness maximum throughput on the DRAM bus, throughput on this bus determines the possible throughput of the system.

---

[1] Recent works [1], [9], in IPv4 forwarding on FPGA achieve 224-673ns address lookup time for table sizes of up to 10,000 entries, which is equivalent to 28-84 cycles at 125MHz.

Latencies were measured assuming a store-and-forward scenario, starting on the cycle when the entire packet arrives at an input, and ending on the cycle the packet starts transmission at an output. There are best case and worst case latencies due to the contention of multiple control streams at different points in the architecture.

Table 2 shows the design statistics for the two and four port routers. We observe that the two-port router is able to achieve a DRAM bus throughput of 1.94 Gbit/s, whereas the four-port router is DRAM limited in terms of throughput to 2.91 Gbit/s (read and write), or 5.82 Gbit/s DRAM bus throughput, 98.5% of the limit derived from MPMC characterization. The maximum two-port memory bus throughput is necessarily less than 2 Gbit/s because only packet payloads pass through the DRAM bus.

TABLE 2
RESULTS FOR THE EXTENSIBLE INTERFACE

	2-Port Router	4-Port Router
Throughput (Gbit/s)	1.94	2.91
Area (slices)	6817	12629
Best case latency (cycles)	87	93
Worst case latency (cycles)	301*	722*

* computed assuming all stages of arbitration hit worst case

The two port router used 6817 slices and the four port router used 12629 slices. Other than the MPMC, which synthesizes to 1318 slices, all logic resources are doubled as we scale from two ports to four ports. The observed increase in area usage confirms that logic resource usage scales linearly. Note however that the implementations were not optimized for area.

1) MPMC latency analysis: The majority of the latency experienced by a packet in this router is before various arbiters. There are five points of arbitration in this router, listed in Table 3. The best case shows shortest latency measured from RTL simulation waveforms. Worst case latency shows longest latencies measured, which include contention at the points of arbitration.

The goal of the example is to stress test the memory sub-system, so a simple internal routing scheme sufficed. As discussed in Section 5.2, there is a 128-cycle period after the header is received to accumulate enough data for the first burst access. The header processing unit can utilize this time to work on a variety of lookup approaches before the first data burst has to be directed to its memory location. The header processing arbitration (1) has negligible latency. The input payload and output payload arbitration delays (2,5) vary dramatically between the best and worst case. The worst case occurs when a burst becomes ready just after its turn at the arbiter. Since four input/output streams are contending for two write/read ports, the latency is approximately two times the worst case

latency of write/read. The best cases for write and read DRAM bus arbitration delay (3,4) are the same as those measured for a single burst. The worst case latency for write and read DRAM bus arbitration however is close to the theoretical latency for four (32-word burst) accesses, where a write/read just misses one round of arbitration, and has to wait for the other three ports to complete before proceeding. The overall latency of one burst comprises the total latency a packet experiences from input port to the output port.

TABLE III
MEASURED LATENCY (4-PORT ROUTER)

	Best Case (cycles)	Worst Case (cycles)
1. Header processing arbitration	2	18
2. Input payload arbitration	2	248
3. Write DRAM bus arbitration	43	135
4. Read DRAM bus arbitration	27	133
5. Output payload arbitration	3	172
Other FSM handshake delays	16	16
Overall latency for one burst	93	722*

* computed assuming all stages of arbitration hit worst case)

In the next section, we investigate and analyze latency reduction techniques that enable reduction in latency to provide a graceful degradation for the worst case as well as improve the average latency for the whole system.

VI. LATENCY REDUCTION TECHNIQUES

Given the many sources of latencies, we propose some latency reduction techniques for latency sensitive packets. We define latency sensitive packets to be those packets that, when routed with shorter latencies, will increase the overall throughput. Specifically, these are packets whose destination port is idling. Reducing the latency of these packets will reduce idle time of the router as a system, thus increasing throughput.

There are two techniques that, applied together, create a solution for the latency problem. The first technique is to create a set of four forwarding paths at the MPMC level. Recall that the base MPMC version we use has two input and two output ports. Input data could be directly sent to output port, bypassing contention on the DRAM bus. Note that these are forwarding paths within the MPMC. Thus the overhead of implementing a fully connected internal path to MPMC is much less than a fully connected cross bar for applications with larger port count. This structure can support arbitrary number of input and output ports on the extensible interface. Figure 5 presents a pictorial representation of these bypasses.

In activating the forwarding path, the sequential order of packets in the output port should be conserved. The forwarding paths are used when the corresponding output queue in the DRAM is empty, where in the absence of the forwarding path, the data has to be written into the DRAM and immediately read out again. The forwarding path reduces the latency to move the data from input port to output port, and conserves DRAM bus time slots. For example, in the above-described 32-word burst case, the forwarding path reduces the 70 to 268 cycle latency (write + read) down to 16 cycles. This approach succeeds at alleviating the contention for the DRAM bus.

However, there remain latencies at the input and output arbiters. With the fast forwarding paths available, there now exist two ways to transfer data from input to output: one writes through the DRAM, and the other bypasses the DRAM.

The main extension here is to allow the input and output arbiters in the extensible interface to prioritize for latency sensitive packets in resolving contentions for gaining access to MPMC port. For scheduling input and output port access, there is a wide range of scheduling algorithms to experiment with: weighted round robin, deficit round robin, weighted fair queuing, etc. In this work we use a non-preemptive round robin scheduling algorithm.

In setting up a fast forwarding operation, the input arbiter needs to a) recognize an empty output queue in memory, and b) wait to synchronize with the output arbiter to transfer the data on the forwarding path. Synchronizing with the output arbiter under-utilizes the input arbiter, and may degrade total throughput. We monitor the queues at the output ports for presence or absence of packets. Based on this information, payloads are routed either through the DRAM memory or using the forwarding paths within the MPMC, bypassing the DRAM memory.

The second technique is to insert a FIFO buffer for a single burst in the forwarding path. The input arbiter is only required to recognize an empty output queue in memory and deposit a burst of data in the buffer for that port. If the output arbiter is busy with other read operations, it can get to the buffered data at the next arbitration opportunity. Since the buffer is a FIFO, the write and read operations can overlap without undesirable side effects. The buffers in Figure 5 represent the above described buffers in the bypass path.

A. Analysis of Latency Reduction Techniques

The bypass path provides a high throughput, low latency alternative compared to accesses through the DRAM. However, it is only activated under particular conditions. For certain traffic patterns, all packets may be able to flow through the bypass path. For some other traffic patterns, the
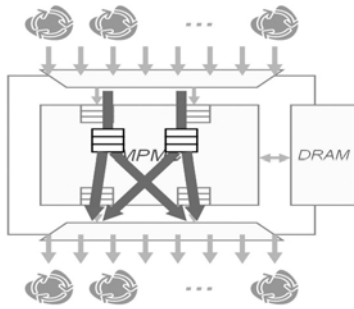
Fig. 5. Latency reduction technique

bypass path may not be used at steady state.

The bypass paths consist of two 64-bit channels that support concurrent read and write. At a 125MHz clock rate, this provides a raw throughput of 16 Gbit/s. Our implementation uses the fast bypass channels when the limited on-chip output queues are sufficient. When the on-chip output queue is exhausted, to avoid HOL blocking (when packets on all input ports are destined to the same output port), the remaining payload is shunted to output queues implemented in off-chip DRAM. The on-chip output queue has 4Kb storage per output port, which holds two to three large-sized standard Ethernet packets (not jumbo grams). The queues are easily exceeded in normal traffic when multiple packets from different sources are destined for the same output port.

As packets are shunted to the DRAM, total throughput over the DRAM bus peaks at 2.9 Gb/s. This may seem to be a sub-optimal steady-state solution where all packets pass through the DRAM interface. However, the output queues are draining at 4 Gb/s. Our system level goal is to increase throughput and reduce idle time at the output ports. We can bias the arbiters of the MPMC to favor pop (read) more than push (write). This way, output queues in the off-chip memory are drained faster than they are filled. Once an output queue in off-chip memory is empty, the bypass path can be utilized again.

We showed that, at steady-state, the bypass paths will contribute positively to the throughput of the router. The total throughput of the four port router stays between 2.9 Gb/s and 4 Gb/s depending on traffic patterns. The latency reduction techniques allow the peak throughput of 4Gb/s. The DRAM interface provides graceful degradation for system throughput down to a minimum of 2.9 Gb/s.

With bypass paths in the latency reduction techniques, we reduced latency from a best case of 70 cycles to as little as 16 cycles, and gained additional throughput in the memory interface. Note that the bypass only affects latency due to write and read DRAM bus arbitration delays (e.g. for the four port router, rows 3 and 4 from Table 3). Prioritizing latency sensitive packets in resolving contentions in input and output arbiter help reduce wait time at row 2 and 5. We emphasize again that the latency reduction here is mainly enabling the additional throughput we are able to achieve and provides a graceful degradation for the worst case scenario.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we have studied issues in building a flexible and scalable interface for external memory controllers in an application domain. We have implemented an interface that builds layers of abstraction starting from a single channel memory controller to an arbitrary ported memory interface. We have presented latency numbers at various stages of the DRAM access pipeline based on our IP routing application. We have presented latency reduction techniques that provide a graceful degradation for worst case scenarios. Lastly, we believe that such extensible interfaces are critical to enabling deployment of component based domain specific languages such as Click. This remains a topic of current research.

REFERENCE

[1] C. Kulkarni, G. Brebner, et. al., "Mapping a domain specific language to a platform FPGA," in Proc. of Design Automation Conference (DAC 2004), San Diego, CA, 2004.

[2] S.A.McKee, W.A. Wulf, et. al., "Dynamic access orderings for streamed computations," in Proc. of IEEE transactions on computers, Vol. 49, Nov 2000.

[3] S. Rixner, W.J. Dally, et. al., "Memory access scheduling," in Proc. of International Symposium on Computer Architecture (ISCA 2000), June 2000.

[4] J. B. Carter, W. Hsieh, et. al., "Impulse: Building a smarter memory controller," in Proc. of IEEE Symposium on High Performance Computer Architecture (HPCA-5), January 1999.

[5] Xilinx memory corner, http://www.xilinx.com/products/ design_resources/mem_corner/resource/xaw_dram_ddr.htm.

[6] J. Park, P. Diniz, "Synthesis of memory access controller for streamed data applications for FPGA-based computing engines," In Proc. of International Symposium on System Synthesis (ISSS'2001), Oct. 2001, pp. 221-226.

[7] S. Iyer, N. McKeown, "Analysis of the parallel packet switch architecture," in IEEE Transactions on Networking, Vol. 11, No. 2, April 2003.

[8] Xilinx application note, "Multi-port memory controller," No. 535, Version 1.1, 2004. (available online)

[9] A. Mihal, K. Keutzer, "A Processing Element and Programming Methodology for Click Elements," Workshop on Application Specific Processors, 10-17, September, 2005.

[10] T. Sherwood , G. Varghese, et. al., "A pipelined memory architecture for high throughput network processors," in Proc. of International Symposium on Computer Architecture (ISCA 2003), June 2003.

[11] J. Hasan, S. Chandra, et. al., "Efficient use of memory bandwidth to improve network processor throughput," In Proc. of International Symposium on Computer Architecture (ISCA 2003), June 2003