HCMA: Supporting High Concurrency of Memory Accesses with Scratchpad Memory in FPGAs

Yangyang Zhao
Institute of Computing Technology
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
zhaoyangyang@ict.ac.cn

Yuhang Liu
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
liuyuhang@ict.ac.cn

Wei Li
Institute of Computing Technology
Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
liwei2012@ict.ac.cn

Mingyu Chen
Institute of Computing Technology
Chinese Academy of Sciences
University of Chinese Academy of Sciences
PengCheng Laboratory
Beijing, China
cmy@ict.ac.cn

***Abstract*—Currently many researches focus on new methods of accelerating memory accesses between memory controller and memory modules. However, the absence of an accelerator for memory accesses between CPU and memory controller wastes the performance benefits of new methods. Therefore, we propose a coordinated batch method to support high concurrency of memory accesses (HCMA). Compared to the conventional method of holding outstanding memory access requests in miss status handling registers (MSHRs), HCMA method takes advantage of scratchpad memory in FPGAs or SoCs to circumvent the limitation of MSHR entries. The concurrency of requests is only limited by the capacity of scratchpad memory. Moreover, to avoid the higher latency when searching more entries, we design an efficient coordinating mechanism based on circular queues.**

We evaluate the performance of HCMA method on an MPSoC FPGA platform. Compared to conventional methods based on MSHRs, HCMA method supports ten times of concurrent memory accesses (from 10 to 128 entries on our evaluation platform). HCMA method achieves up to 2.72× memory bandwidth utilization for applications that access memory with massive fine-grained random requests, and to 3.46× memory bandwidth utilization for stream-based memory accesses. For real applications like CG, our method improves speedup performance by 29.87%.

***Index Terms*—memory request accelerator, coordinated batch method, circular queue, FPGA**

I. Introduction

Nowadays the conventional memory controller uses standard DDRx SDRAM based bus interface. To overcome the limitations of fixed delay and granularity, researchers focus on asynchronous and parallel memory bus based on batch processing. Typically intermediate logic, such as an interface buffer/die, is introduced to bridge processor and memory. This new architecture includes FBDIMM [1], BOB [2], and HMC [3]. Other solutions like Gen-Z [4] and MIMS [5] attempt to create a new bus protocol.

However, the design of new architecture merely solves part of the problem in transferring requests for highly concurrent memory accesses. Current cache hierarchies have limited miss status handling registers (MSHRs) to support request parallelism between CPU and memory controller. Even in designs for high-end commercial processors, the norm supports only a very modest number of outstanding misses at a time [6]. For embedded processors and MPSoC FPGAs, the MSHR limitation issues are more severe [7] [8] since the MSHR numbers are relatively small (10 in Xilinx Zynq-7000 all programmable SoC) but the memory access requirements are normally more critical (benefits from new architectures). Due to the limited physical resources available on chip, increasing the number of MSHRs can be challenging.

We present a general method to achieve high concurrency of memory accesses (HCMA) between CPU and memory controller practically. Without modifying the CPU cores, we propose a coordinated batch method based on scratchpad memory to replace the conventional method using MSHRs. Moreover, we design an efficient coordinating mechanism based on circular queues to manage the states of all records. Furthermore, to take advantage of the performance benefits of new architectures, the design supports basic packing and unpacking process [9], and some extended memory access instructions, such as scatter/gather [10].

We implemented HCMA on Xilinx Zynq series MPSoC FPGA platform. Experiment results show that compared with conventional memory accesses, HCMA module with extended memory access instructions presents significant performance improvements when running applications with different memory access features. Random access [11] test obtains 2.72× memory bandwidth utilization, while the result of stream access [12] test is 3.46× memory bandwidth utilization. For real applications like CG [13], the performance is increased by 29.87%.

The key contributions of our study are as follows:

1) We propose a general SPM-based coordinating method to manipulate highly concurrent memory requests in batch without altering CPU cores and the NoC protocol: HCMA method communicates with CPU cores via scratchpad memory (SPM). Memory requests are sent to SPM by software. Then HCMA related modules fetch the requests from SPM. As for conventional methods, the amount of outstanding memory accesses is limited by the number of MSHRs. HCMA method holds requests with SPM, which eliminates the concurrency restrictions incurred by MSHRs. The amount of storage entries can easily exceed ten times the number of MSHR entries.

2) We propose extended memory access instructions that take advantage of the new architecture for concurrent memory accesses between memory controller and memory modules: By utilizing SPM, HCMA method defines extended memory access instructions without modifications of CPU cores. As an example, we designed scatter/gather instruction for higher memory bandwidth, since it removes the limitation of fixed data granularity.

3) We implemented a practical HCMA module on an MP-SoC FPGA platform integrated with SPM and evaluated the performance improvements: We implemented HCMA module with programming logic on FPGA platform to improve the concurrency of memory requests. Experiment results showed that the design provides better performance in memory bandwidth than conventional methods.

This paper is organized as follows: Section II presents background information of our study. In Section III, we provide general coordinated batch method based on SPM. Section IV details the implementation of HCMA method on FPGA platform and lists the amount of resources consumed by HCMA module. In Section V, we introduce the parameters set for experiments, the features of application programs, and then analyze the results of performance tests. Section VI discusses the scalability of HCMA method. Section VII summarizes the paper.

II. Background

The conventional memory architecture adds multi-level caches between CPU and memory, as shown in Figure 1(a). This architecture masks the memory access delay by increasing cache hit rates, while relieves the memory wall problem by improving off-chip memory bandwidth. However, it lacks efficiency and scalability in the new scenario that applications send high concurrency requests. First, when using the standard DDRx memory interface, the processor and the memory are tight coupling. Therefore, the delay of data access is fixed, which increases the difficulty of sending highly concurrent requests. Second, the mismatch between the fixed granularity of data transmission and the variable data sizes of different applications causes the waste of memory bandwidth.

The improved methods based on this architecture mainly focus on the following research aspects. The most effective

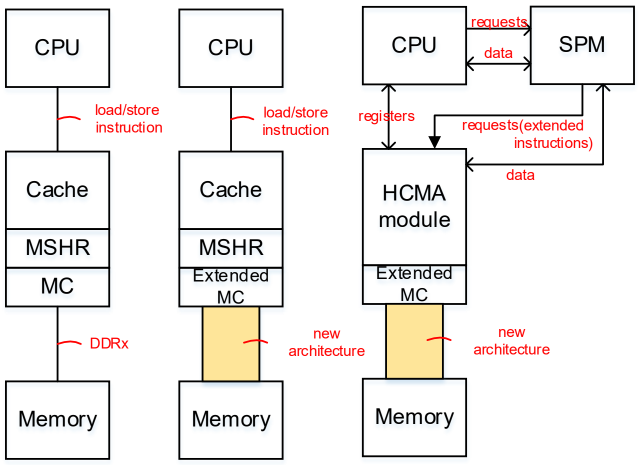


Fig. 1. The architecture design of memory access

way to increase cache hit rates is to achieve good temporal and spatial locality [14] [15], which is becoming more and more difficult to reach with multicore processors and data-intensive applications nowadays [16] [17]. At the same time, the most natural and classical way to increase memory bandwidth is to increase the memory clock frequency [18] [19] or the memory bus width [20] [21]. However, both parameters are close to the physical limits of traditional memory like DRAM [22].

Therefore, the study of new architecture with asynchronous interface, variable granularity, and variable delay has become a trend, including BOB [2], HMC [3], Centaur [23], and Gen-Z [4]. The new architecture transfers packages in asynchronous request and response modes, therefore a single transfer possibly supports multiple concurrent and out-of-order requests. The new architecture is shown in Figure 1(b). The basic idea of new architecture is to add buffer chip to the system for building an extended memory controller, and then make use of new high-speed bus interface to connect extended memory. An example of new architecture is MIMS [5], which is a specific design of BOB [2]. A message channel in MIMS replaces the conventional synchronous memory bus. The message channel works in request and response modes, which transfer message packages with asynchronous memory accesses. Moreover, MIMS supports sending multiple concurrent and out-of-order requests within a single message packet. Therefore, the memory bandwidth can be well utilized. In addition, memory modules are organized and managed by their own buffer and scheduler instead of the processor. MIMS further improves memory access performance with cooperation between the processor and the memory system.

Current researches mainly focus on the interface between memory controller and memory modules. The optimization goal is to improve bus parallelism. The requests transfer between CPU cores and memory controller still uses conventional methods, as shown in Figure 1(b). If cache hits, CPU accesses cache according to the memory address translation information. If cache misses, MSHRs record outstanding requests. Unfortunately, in current architecture, the number of MSHRs is quite limited especially in embedded SoC(10 outstanding read requests on ZC706 board [24]). When an application with poor locality runs, MSHRs will be filled full quickly and become the bottleneck of highly concurrent applications even with new architecture. The problem above cannot be solved by simply extending the number of MSHRs [6]. Considering the mismatch between request granularity(4KB, 2MB, etc.) and memory access granularity(32B, 64B, etc.), one request is probably recorded in multiple MSHR entries. In addition, it is difficult to add sufficient amounts of MSHRs limited by on-chip resources.

To break the limit of concurrency, we design a request batch processing method based on scratchpad memory. The structure is shown in Figure 1(c). The scratchpad memory is addressable by CPU. CPU writes memory access requests in batch to different addresses in SPM. HCMA module reads memory access requests in batch from SPM and executes the requests. If the requests are memory read requests, after the data is returned from memory, HCMA module puts data into SPM and informs CPU to read data. If the requests are memory write requests, there is no need to return data. Therefore, HCMA module only needs to execute the requests.

Our approach replaces cache layers and MSHRs with SPM. In principle, the absence of cache will introduce delay in accessing data. However, in the case of highly concurrent memory requests, MSHRs will fill full quickly and then lock up the cache, causing a long delay. On the contrary, the SPM-based method stores more unfinished requests and multiple requests execute concurrently. In high concurrency scenarios, the performance of SPM-based method for batch processing is not affected, although the delay of a single request will be longer than traditional method.

HCMA method is practical since there is no need to alter CPU cores and the NoC protocol. The only precondition is that the on-chip system is integrated with scratchpad memory, which can be accessed by both CPU and FPGA logic. Then the method can be generally used for both FPGA memory expansion and dedicated SoC design.

III. Design of HCMA

As shown in Figure 1(c), the read and write requests are transferred through SPM from CPU to HCMA module, while the transmission of data is also accomplished via SPM. Therefore, the first problem in our design is the communication between CPU and HCMA module. We need to design a method to complete read and write requests with a reasonable process flow. Since the design significantly increases storage entries to circumvent the limitation of MSHR entries, the management problem of multiple entries must be addressed. Moreover, in order to make full use of extended memory controller, the HCMA method needs to add extended instructions. The following section will describe the three aspects of HCMA method, including SPM-based request processing flow, management mechanism with circular queue and design of extended memory access instructions.

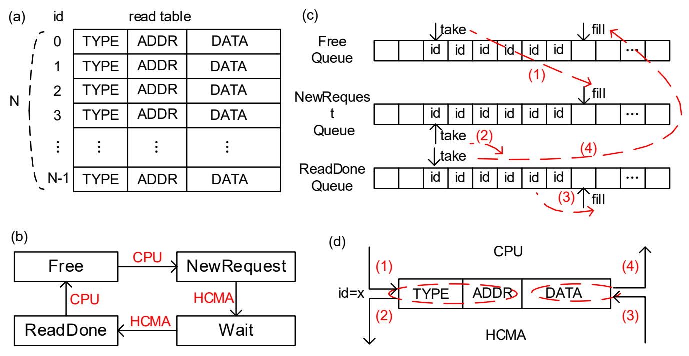


Fig. 2. The SPM-based batch processing for read requests

A. SPM-based Request Processing

We use SPM as the reserved space to transfer requests between CPU and HCMA module.

1) Read Request Management:

As shown in Figure 2(a), SPM stores one read table. A read table can store a lot of entries. Each entry corresponds to an ID number and stores a single read request. Each read request occupies a fixed-length address space with three fields: TYPE, ADDR and DATA, which are used to store the type, address and data of the read request separately.

In Figure 2(b), each entry in the table changes in four states: Free, NewRequest, Wait, ReadDone. In 'Free' mode, there is no read request. When a read request is sent by CPU cores, the state of corresponding entry changes to 'NewRequest', which represents new read request is received. In 'NewRequest' mode, HCMA fetches read requests from SPM and sends the requests to extended memory. Then the state transforms from 'NewRequest' to 'Wait'. After a read request is completed, which means HCMA has received the requested data and the data has been filled into the 'DATA' field, the state changes to 'ReadDone'. After CPU cores obtain the required data, the state of corresponding entry changes to 'Free'. Then the entry is ready for another read request processing.

To manage the states of multiple entries (the number is N in Figure 2(a)) in the table, we developed a management method based on circular queues, which provides rapid detection of the state in the table.

We provide three circular queues for different states of table. They are named as 'Free Queue', 'NewRequest Queue', 'ReadDone Queue'. As shown in Figure 2(c), all the queues are used to store the ID value of entries which match the corresponding state. Since the match between returned data and request is judged by the ID value, the three queues can be managed out of order, which means the method supports out-of-order execution. Each queue has both take pointer and fill pointer, which indicate head position and tail position respectively. The take and fill pointers drawn above the queues are used by CPU and the take and fill pointers drawn below the queues are used by HCMA. All pointers are implemented by FPGA registers, so that HCMA module can detect changes rapidly. CPU updates pointers by accessing corresponding memory-mapped space. CPU uses methods like polling to detect the updates of registers. Therefore, the communication of registers is not related to SPM, as shown in Figure1(c). The queues are implemented by arrays. In principle, it is reasonable to take the total number of entries as the length of each array, since in the extreme situation all the entries are in the same state, then one of the three queues are filled full with ID numbers. However, to avoid the comparison of take and fill pointers when adding IDs to the queue, we design the length of each array to be the total number of entries plus one, i.e. the length of each queue is N+1 in Figure 2(c), then we ensure all the queues will never be full.

The batch processing flow for read requests makes use of the three circular queues. The read flow is divided into the following four steps, as shown in Figure 2(c) and Figure 2(d):

- First, the software check whether the 'Free Queue' in Figure 2(c) is empty (a queue is empty when the take pointer equals to the fill pointer). If not, take some IDs from the location pointed by the take pointer and fill them to the location pointed by the fill pointer of 'NewRequest Queue'. At the same time, the table entries related to the IDs are filled with new read requests, Figure 2(d) takes one entry(ID equals to X) as an example, two fields (TYPE, ADDR) are filled.
- HCMA reads the IDs from the take pointer of the 'NewRequest Queue' in Figure 2(c), then takes the read requests according to the ID values and sends them to extended memory. Figure 2(d) takes one entry as an example, HCMA sends the contents of two fields (TYPE, ADDR).
- After the read request data returns, HCMA fills in the 'DATA' field of the corresponding read table entry as shown in Figure 2(d), meanwhile it places the ID in the position indicated by the fill pointer in the 'ReadDone Queue' in Figure 2(c).
- The software takes the IDs from the take pointer in the 'ReadDone Queue', and then fetches the data from the table to the CPU according to the IDs, and finally puts the IDs into the 'Free Queue'.

With the SPM-based batch processing method, the numbers of table entry is only limited by the capacity of on-chip memory. The concurrency exceeds the traditional method obviously.

2) Write Request Management:

The circular queue used to transfer write requests is shown in Figure 3. There is no need for write requests to return data like read requests, therefore the write flow is simplified as follows:

- When to send new write requests, the software checks whether the write queue is full. If the queue is not full, the software fill the corresponding location pointed by the fill pointer with new write requests.
- HCMA takes the write requests from the location correlated to the take pointer of the write queue, and sends them to extended memory.

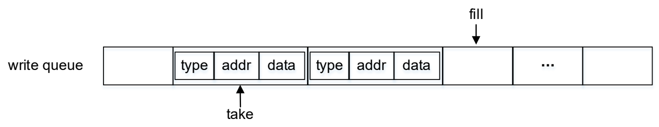


Fig. 3. The SPM-based batch processing for write requests

The difference between read and write request processing is that the write queue is possible to be full. Therefore, the first step is to check if the queue is full when sending new requests.

B. Extended Memory Access Instructions

As shown in Figure 2(d) and Figure 3, there exists a 'TYPE' field for both read and write requests. This feature is used for the design of extended memory access instructions. Moreover, the 'TYPE' field can carry some upper-level information, such as thread number and priority. These parameters are used to achieve advanced QoS scheduling in extended memory.

1) Variable Granularity:

The granularity information is encoded in the 'TYPE' field, which is not limited by the instruction set and cache line size. Therefore, 8B to 4KB memory access granularity can be sent by software. The software control the granularity of memory accesses directly, therefore the memory optimization can use some advanced memory access strategies at the software level. In addition, the DRAM accesses are all needed by software, achieving effective memory bandwidth.

2) Scatter/Gather:

Scatter/gather is a method to implement DMA data transfer [25]. It uses a link list to describe physical discontinuous addresses of memory access. CPU transfers the first address of link list to DMA, and then DMA master transmits a continuous block. After all the data from discontinuous addresses is delivered, the method generates an interrupt. Scatter/gather is considered as an efficient DMA approach. In the design, the user-defined feature of 'TYPE' field in SPM offers the possibility to design extended memory access instructions, such as scatter/gather. Scatter/gather data transfer is as follows: First, the method reserves storage space for scatter/gather transmission in SPM. When the 'TYPE' field is set to be a scatter/gather instruction, the software writes multiple requests to the storage space. The HCMA module detects the changes of related status register, and then fetches a request from SPM. If the 'TYPE' field of the request matches scatter/gather instruction, HCMA module obtains all other requests of the extended instruction, while processing the requests in batch. After all the requests are finished, HCMA module updates the value of related status register. The advantage of this design is that it reduces the number of accesses to SPM by CPU and HCMA module, further improving the performance of request transfer.

IV. Implementation of HCMA

The verification platform needs to meet the following requirements:

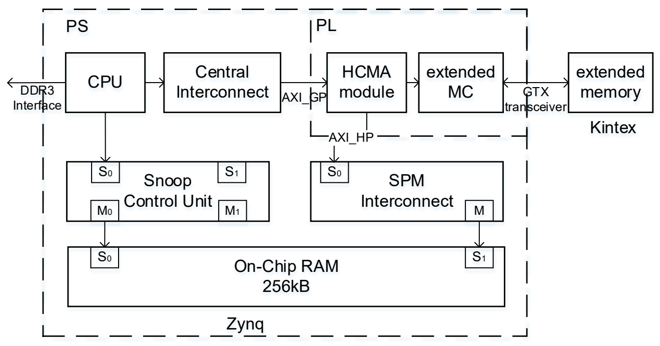


Fig. 4. The system interconnect architecture with HCMA

- To achieve request processing with HCMA, the FPGA platform requires CPU cores, scratchpad memory, and an interface between FPGA logic and SPM. MPSoC FPGA (XC7Z045) belongs to the Zynq-7000 all programmable SoCs, referred to as Zynq in Figure 4. It integrates an ARM-based processor system with FPGA [24]. The architecture can customize both logic and software.
- To realize asynchronous memory access with extended memory, it is necessary to simulate asynchronous new architecture. Another Kintex chip(XC7K410T) is used to analog the receiver of new architecture, and connected to Zynq through GTX (Giga bit TX) high speed interface.
- To compare with conventional methods, we reserved the interface between Zynq and DDR3 SDRAM as a baseline.

In the following part, we will describe the features of the MPSoC FPGA platform. Then we will detail the implementations of HCMA, including the system interconnect architecture, the processing flow of memory access requests, and the circular queue management mechanism to replace polling method.

A. System Interconnect Architecture

PS is the abbreviation of processing system. As shown in Figure 4, PS contains a 256KB scratchpad memory, also known as OCM. It uses SRAM circuit. Therefore, the speed of OCM access is much faster than DRAM main memory access. OCM occupies specific physical address space. CPU sends read and write requests via the snoop control unit. Meanwhile, HCMA module accesses OCM via the OCM interconnect unit as a master.

HCMA is implemented in PL, which means the programming logic. The interconnect bus between PS and PL is composed by three kinds of AXI (advanced extensible interface) bus. Accelerator-Coherency Port, referred to as AXI_ACP, transfers requests that ensuring cache consistency. General-Purpose Port, referred to as AXI_GP, is suitable to connect I/O devices. High-Performance Port, referred to as AXI_HP, supports data transfer with a large size. In the design, we use AXI_GP to transfer memory access requests from CPU to HCMA module, while the AXI_HP transfers data between HCMA module and OCM.

The extended memory controller adds semantic information to memory access requests, encapsulates multiple requests into a packet, and then transfers packets to the extended memory. The extended memory consists of buffer chip and memory. We use high-speed Xilinx GTX transceiver to accomplish the data transfer between extended memory controller and the extended memory.

Moreover, we reserved the conventional interface between Zynq and DDR3 SDRAM as the baseline of the experiment, as shown in Figure 4. The baseline memory access is sent to DRAM memory (1GB) via L1 and L2 cache.

B. HCMA Internal Modules

There are two submodules in HCMA, namely read forwarder and write forwarder. The outputs of read forwarder and write forwarder are passed to request queues that implemented by FIFOs. In addition, the internal response queue that holds the IDs of finished read requests is also implemented by FIFO. HCMA also contains two sets of registers. One is named the group of base address registers, while the other is the group of pointer registers. In Zynq, CPU can access the registers in HCMA via AXI_GP.

Base address registers are written by CPU at the beginning of the program. Then they will be used by HCMA when the program is running. The values of them remain the same during program execution. The registers include the base address of table, new read queue, finished read queue and write queue. The base address of table is used by read forwarder in HCMA to calculate the OCM address of current read request. OCM address equals to the sum of table base address and ID value. When pointers reach the end of arrays, they use the base address of new read queue, finished read queue and write queue to return to the beginning of the arrays. This circular feature of HCMA method ensures the communication with shared scratchpad memory.

Pointer registers are divided into two parts. One is frequently updated by CPU when program is running, while the other is updated by HCMA. The former includes the fill pointer registers of new read queue and write queue. HCMA submodules check these registers for the number of elements in queues and determine whether to fetch requests from OCM according to the values of the registers. The later includes the fill pointer register of finished read queue and the take pointer register of write queue. CPU uses the fill pointer to determine which read request has been completed, and checks the take pointer to estimate if write queue is full. The design of pointers supports out-of-order and parallel execution of requests.

Table I lists the amount of resources consumed by the different components of HCMA module. RAMB36E1 and RAMB18E1 represent 36Kb and 18Kb of Block RAM, respectively. We make use of multiple FIFOs in PL, which occupy some block RAM resources. Meanwhile, when calculating the access address of the table and the queues, multiplication is used, so the design occupies four DSPs, i.e. DSP48E1 in the table.

TABLE I
RESOURCE UTILIZATION

category	quantity	Total	Proportion
Register	8697	437200	1.99%
LUT	5792	218600	2.65%
Slice	2831	54650	5.18%
RAMB36E1	76	545	13.94%
RAMB18E1	3	1090	0.28%
DSP48E1	4	900	0.44%
BUFG	3	32	9.38%

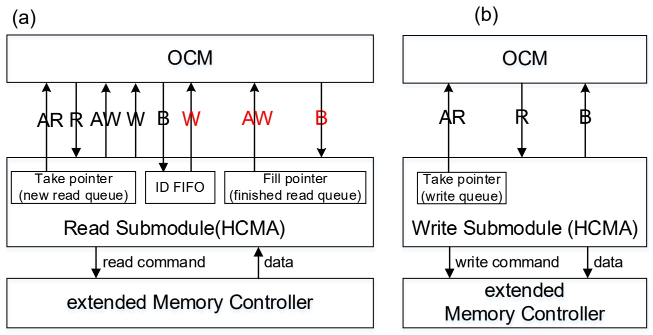
C. CPU-FPGA Communication

After configuring the programming logic, the complete process for HCMA to transfer read and write requests in batch is shown in Figure 5. The read request flow is as follows:

- First, read forwarder of HCMA module uses AR channel of AXI_HP0 to send read command to OCM, the address is corresponding to the take pointer of 'NewRequest queue'. OCM then executes this read command, returning read requests in the queue to read forwarder via R channel of AXI_HP0.
- Second, read forwarder analyzes the read request, and it sends a read command to the extended memory controller and record ID of the read request. The address is in the 'addr' field of the read request. Extended memory controller will execute this read command, after the execution, the data will be returned.
- The read forwarder then transfers data to the table in OCM through AW and W channels of AXI_HP0. The address of the AW channel is determined by the ID returned.
- When data transfer completes, OCM sends a 'write response' to read forwarder through B channel of AX-I_HP1. The forwarder places the ID of B channel into the ID FIFO of the forwarder.
- When the number of data in the ID FIFO reaches a certain amount, the forwarder writes the IDs to the finished read queue of OCM through AW and W channels of AXI_HP1. The address of AW channel is the fill pointer of the finished read queue. When the transfer completes, OCM sends a 'write response' to read forwarder through B channel of AXI_HP1.

Figure 5(b) shows the write request flow. First, write forwarder sends read commands to OCM through AR channel of AXI_HP2, and then OCM executes this read command, returning the data in write queue to the forwarder through the R channel of AXI_HP2.

Then write forwarder analyzes the data, it sends a write command to the extended memory controller. The address is from the 'addr' field in the write request, and the data is from the 'data' field in the write request. After the execution, the write forwarder will send a 'write response' to OCM through the B channel of AXI_HP2.



(a)
OCM
AR
R
AW
W
B
W
AW
B
Take pointer (new read queue)
ID FIFO
Fill pointer (finished read queue)
Read Submodule(HCMA)
read command
data
extended Memory Controller
(b)
OCM
AR
R
B
Take pointer (write queue)
Write Submodule (HCMA)
write command
data
extended Memory Controller

Fig. 5. The process flow of read submodule and write submodule in HCMA

D. Software Modification

Zynq-based software development is performed in the Xilinx Software Development Kit (referred to as XSDK). XSDK has an ARM gcc cross compiler that compiles software code into an executable file. An SSH connection can be established in the XSDK environment to log in to the Zynq Linux system. When running the software, XSDK will automatically copy the executable file to the Linux file system for execution.

We propose HCMA without altering CPU cores and the NoC protocol. To implement HCMA using SPM, the software codes are modified as follows.

First, software codes use Linux system call mmap() to complete the conversion of virtual and physical addresses. Because the addresses stored in the table and write queue are physical addresses, while the software uses virtual addresses.

Second, the codes define new data structures for HCMA, the sizes and address offsets of all data structures in the OCM. The data structures include 'read table', 'Free Queue', 'NewRequest Queue', 'ReadDone Queue' and 'write queue'. The codes also writes all IDs into the 'Free Queue'.

Then, according to the read and write flow described in Section III, we write algorithms that implements HCMA read and write requests. It is worth mentioning that the algorithms are all about read and write OCM space and update registers. When applications need to send high concurrent read or write requests, they call our customized HCMA_READ or HCMA_WRITE function.

Finally the modified codes can be compiled to an executable file to test.

V. EVALUATIONS

A. Methodology

Our verification platform supports two kinds of memory accesses. The first kind is to transfer requests to standard DDR3 memory via DDR3 memory bus, as shown in Figure 4. Requests access DDR3 SDRAM when L2 cache miss occurs. This kind is presented as the baseline of the evaluation. The other kind is to transfer requests to extended memory controller in programming logic. Then extended memory controller communicates with extended memory to fetch data. We evaluate the performance of both kinds.

TABLE II
BASELINE, OCM, HCMA RUNNING CONDITIONS

	Baseline	OCM	HCMA
data width (bits)	32	64	64
frequency (MHz)	533	222	100
physical limited peak bandwidth (GB/s)	1.066	1.776	0.8

The operating parameters for baseline memory, OCM, and extended memory are shown in Table II. As listed in Table II, both data width and the physical limited peak bandwidth are different for the three kinds of memory. To ensure the fairness of evaluation, it is better to take bandwidth utilization as the indicator. The bandwidth utilization equals to the results of dividing practical memory bandwidth by physical limited peak bandwidth.

B. Performance

1) Random Access:

Random access is a program used by HPC Challenge Benchmark to evaluate memory random access performance. It measures how many times the CPU cores can update the randomly generated memory address per second. Random access attempts to issue as many random update operations as possible to memory system, in order to get the peak performance. Therefore, it is a typical program with highly concurrent requests. Theoretically the high concurrency mode will achieve better performance.

The test results of random access are measured with GUPS (Giga UPdates per Second). The granularity for each access is 8 bytes. An update means a read access and a write access. Therefore, the memory bandwidth is 16 times the value of GUPS. Figure 6 shows the memory bandwidth in four conditions: baseline, HCMA module, the on-chip memory, HCMA module with extended instructions. It can be seen from Figure 6 that the results of HCMA method are better than baseline.

The memory bandwidth utilization is the ratio of bandwidth in test and bandwidth in physical limited peak bandwidth. The value of baseline is only 7.6% (0.0811/1.066). This result highlights the performance bottleneck of the conventional memory when facing highly concurrent memory access. The memory bandwidth utilization of HCMA is 13.6% (0.1089/0.8), the advantage is not obvious as expected. We analyze the reason as follows. To support high concurrency, HCMA method needs to read and write OCM more than four times to finish one memory access request. So the performance of OCM access must be fast enough, otherwise it will become the bottleneck and the performance of HCMA method is affected. To verify the hypothesis, we add two other experiments.

First, we run random access on OCM. The difference from other experiments is the table size set for the program. Since OCM only has 256KB capacity, it is impossible to support table size of 16MB, so we choose 16KB in the experiment. Even if the data amount is reduced, the memory bandwidth utilization is only 7.9% (0.1397/1.776). Therefore, we proved

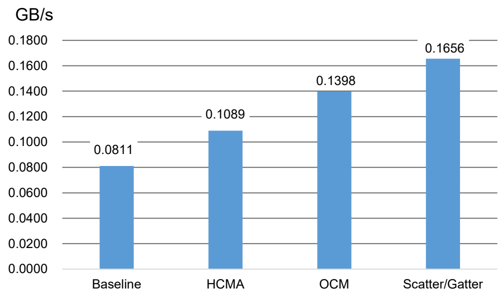


Fig. 6. The memory bandwidth of Random Access

that excessive access to OCM is the bottleneck. The other experiment is to run the program with extended scatter/gather function. To reduce the number of accesses to OCM, we sent eight read or write requests in one scatter/gather instruction. The table size is still set to 16MB. The memory bandwidth utilization is 20.7% (0.1656/0.8). The practical memory bandwidth utilization reaches more than 2 times of baseline.

In summary, HCMA method makes better use of memory bandwidth when applications are highly concurrent. When introduces scatter/gather instruction, the memory bandwidth utilization reaches 2.72$\times$ of baseline. The performance of OCM access in Zynq platform is limited, which weakens the advantage of HCMA method. It is possible to further optimize the results on other advanced platforms, since HCMA method has not reached the ideal peak performance in current platform.

2) Stream Access:

Stream access is a program used by HPC Challenge Benchmark to test memory or storage performance. The processor produces memory access requests with consecutive addresses. The program includes four simple vector calculations: Copy, Scale, Add, and Triad [12]. For stream access test, request addresses are continuous, and the amount of request data is large. HCMA design reserves the ‘type’ field for extended instructions.

We modify the program to transfer memory access requests with 1KB granularity. Figure 7 shows the test results of memory bandwidth utilization both in baseline and HCMA mode. In the Copy test, the bandwidth utilization of high concurrency memory (1.7288/0.8) is 3.46$/times$ of the conventional memory (0.6657/1.066). In the Scale, Add and Triad tests, the advantages of the high concurrency mode are not obvious. The fundamental reason is that all of the tests require frequent communications between CPU cores and OCM, which interfere with the transfer of requests and lead to a corresponding reduction in memory bandwidth. However, the test results are still better than baseline.

3) CG:

The CG benchmark comes from NAS Parallel Benchmarks. It is developed by NASA to evaluate the performance of large-scale parallel computer systems. The CG program will send random and out-of-order access to memory. To evaluate the performance of applications like CG, presenting the speedup

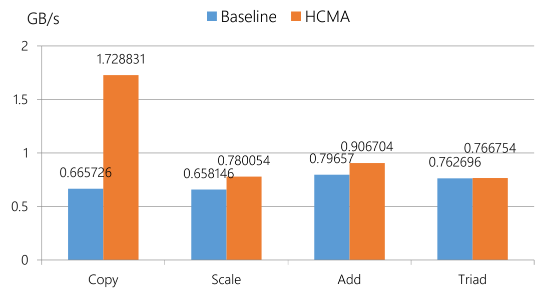


Fig. 7. The memory bandwidth of Stream Access

performance will be more intuitive. Speedup equals to execution time of the baseline divided by execution time of HCMA method. For CG, we get speedup performance increased by 29.87%.

VI. Discussions

HCMA method uses SPM to transfer requests and data and therefore both CPU and FPGA access SPM frequently. To get better performance, the SPM access latency and bandwidth are more concerned than capacity. The two Cortex-A9 processors integrated in our experiment platform have limited ability to send concurrent requests and the speed of on-chip SPM bus is also limited. For future work, we will apply HCMA method to new Zynq Ultra series, which will achieve better performance with more powerful four Cortex-A53 cores. When multi-cores share the memory, only software needs to be modified to ensure the consistency of memory access. The consistency problem can be solved by traditional methods, such as locking.

HCMA method is a general method. We use Zynq platform currently because it is integrated with hard cores to access quickly. Any hardware platforms with cores are available for HCMA method. For example, HCMA method can be implemented with software processor cores in FPGA or other acceleration units that need to access memory, such as PE arrays. HCMA method can also be used in SoC ASIC design.

VII. Conclusions

In this study, we propose an SPM-based mechanism for manipulating highly concurrent memory requests without altering CPU cores and the NoC protocol. HCMA method also introduces extended memory access instructions through memory bus with new architecture. We design and implement HCMA in an MPSoC FPGA platform and compare the performance of applications with conventional method. The evaluation results show that HCMA offers higher performance than baseline. HCMA can be used on MPSoC FPGA platform directly or integrated into future SoC designs. Our future work will focus on advanced platform to support high concurrency.

Acknowledgment

This work is supported by National Key Research and Development Plan of China 2017YFB1001600, NSFC (National Science Foundation of China) No. 61772497 and No. 61521092, State Key Laboratory of Computer Architecture Foundation under Grant No. CARCH2601.

References

[1] B.Ganesh et al. Fully-Buffered DIMM Memory Architectures: Understanding Mechanisms, Overheads and Scaling. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2007.

[2] E.Cooper-Balis, P.Rosenfeld, B.Jacob. Buffer-On-Board Memory System. In *Proceedings of the 39th International Symposium on Computer Architecture(ISCA '12)*, 2012.

[3] J.T.Pawlowski. Hybrid Memory Cube. In *Hot Chips: A Symposium on High Performance Chips*, Aug 2011.

[4] K.Bowman. Gen-Z Technology: Enabling Memory Centric Architecture. In *Flash Memory Summit*, Aug 2017.

[5] L.C.Chen, M.Y.Chen, Y.Ruan, Y.B.Huang, Z.H.Cui, T.Y.Lu, and Y.G.Bao. MIMS Towards: a message interface based memory system. In *Journal of Computer Science and Technology, 2014*, pp.255C272.

[6] J.Tuck, L.Ceze, J.Torrellas. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*,2006. pp.409-422.

[7] Z.M.Fang, S.Mehta, P.C.Yew et al. Measuring Microarchitectural Details of Multi- and Many-Core Memory Systems through Microbenchmarking. In *ACM Transactions on Architecture and Code Optimization(TACO)*, Jan 2015.

[8] P.K.Valsan, H.Yun, F.Farshchi. Addressing isolation challenges of non-blocking caches for multicore real-time systems. In *Real-Time Systems*, Sep 2017.

[9] T.Y.Lu, L.C.Chen, and M.Y.Chen. Achieving effcient packet-based memory system by exploiting correlation of memory requests. In *Proceedings of the conference on Design, Automation & Test in Europe* $DATE$, 2014.

[10] 'Linpack,' http://www.netlib.org/linpack/

[11] 'Randomaccess C gups (giga updates per second),' https://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/, 2012.

[12] 'Stream: Sustainable memory bandwidth in high performance computers,' http://www.cs.virginia.edu/stream/, 2012.

[13] 'Nas parallel benchmarks,' http://www.nas.nasa.gov/publications/npb.html, 2012.

[14] G.Kurian et al. ATAC: A 1000-core cache-coherent processor with on-chip optical network. In *International Conference on Parallel Architectures and Compilation Techniques*, 2017. pp.477-488.

[15] Y.Li, R.Melhem, A.Abousamra, AK.Jones. Compiler-assisted data distribution for chip multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques*, 2017. pp.501-512.

[16] S.G.Ahmad, C.S.Liew, M.M.Rafique, E.U.Munir. Optimization of data-intensive workflows in stream-based data processing models. In *Journal of Supercomputing*, 2017.pp.1-23.

[17] M.H.Moghadam, S.M.Babamir, M.Mirabi. A Multi-Objective Optimization Model for Data-Intensive Workflow Scheduling in Data Grids. In *IEEE 41st Conference on Local Computer Networks Workshopss*, 2016. pp.25-33.

[18] R.Appuswamy et al. Scaling the Memory Power Wall With DRAM-Aware Data Management. In *Proceedings of the 11th International Workshop on Data Management on New Hardware(DaMoN'15)*, 2015.

[19] Q.Zhang et al. CHARM: A Cost-efficient Multi-cloud Data Hosting Scheme with High Availability. In *IEEE Transactions on Cloud Computing*, 2015. pp.372-386.

[20] I.Bhati et al. DRAM Refresh Mechanisms, Penalties, and Trade-Offs. In *IEEE Transactions on Computers*, 2015. pp.108-121.

[21] S.Goossens et al. Power/Performance Trade-offs in Real-Time SDRAM Command Scheduling. In *IEEE Transactions on Computers*, 2016.

[22] 'DDR4 Products,' http://www.montage-tech.com/DDR4/index.html

[23] M.Owaida, D.Sidler, K.Kara, G.Alonso. Centaur: A Framework for Hybrid CPU-FPGA Databases. In *FCCM'17*.

[24] 'ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC,' https://china.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf

[25] K.Nayak et al. GraphSC: Parallel Secure Computation Made Easy. In *IEEE Symposium on Security and Privacy*, 2015. pp.377-394.