Direct Virtual Memory Access from FPGA for High-Productivity Heterogeneous Computing

Ho-Cheung Ng, Yuk-Ming Choi, Hayden Kwok-Hay So
Department of Electrical & Electronic Engineering, The University of Hong Kong
{hcng, ymchoi, hso}@eee.hku.hk

***Abstract*—Heterogeneous computing utilizing both CPU and FPGA requires access to data in the main memory from both devices. While a typical system relies on software executing on the CPU to orchestrate all data movements between the FPGA and the main memory, our demo presents a complementary FPGA-centric approach that allows gateware to directly access the virtual memory space as part of the executing process without involving the CPU. A caching address translation buffer was implemented alongside the user FPGA gateware to provide runtime mapping between virtual and physical memory addresses. The system was implemented on a commercial off-the-shelf FPGA add-on card to demonstrate the viability of such approach in low-cost systems. Experiment demonstrated reasonable performance improvement when compared to a typical software-centric implementation; while the number of context switches between FPGA and CPU in both kernel and user mode was significantly reduced, freeing the CPU for other concurrent user tasks.**

I. Introduction

Memory subsystem is an indispensable part of modern processor systems where both program instructions and data are stored during runtime. In a heterogeneous computer, efficient access to the main memory from both software executing on the CPU and gateware executing on FPGAs is imperative.

Despite its importance, interfacing with the main memory in heterogeneous computers often relies on vendor-specific software libraries and gateware modules that are incompatible with one another. Also, the judicious use of storage elements such as FPGA on-chip and on-board memory or system memory are often left to the application designers to determine. Consequently, results from different researchers are often difficult to compare and reuse, hindering their collaborations.

This demo presents our approach to this problem by adopting a FPGA-centric memory access model, which allows gateware, i.e., the soft-compute engine that resides on the FPGA, to access its own virtual memory address space autonomous to the CPU. This model follows naturally from the autonomous gateware execution model previously proposed by work such as BORPH [1], FUSE [2] and in the Convey HC-1 system [3] as a way to improve designer productivity and tools portability. As the FPGA gateware is capable of addressing the entire memory space, simple zero-data-copy transfers of control between software and gateware can be achieved as shown at the top of Figure 1. This is in contrast to a typical system in which software must manage all data movements even after the control has been transferred to the FPGA.

To integrate such computing model effectively in a concurrent multi-user system, however, requires efficient virtual memory support for the gateware executing on FPGAs. This is particularly important in systems that compute with commercial-off-the-shelf FPGA add-on cards as they reside in the I/O space of the host system with limited connectivity to the main memory.

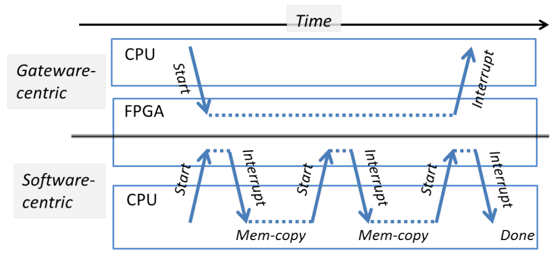

Fig. 1. Comparison between software-centric and gateware-centric memory access models. Typical software-centric approach (bottom) may be highly optimized while gateware-centric approach (top) is simpler but requires direct virtual memory access from gateware.

To address this, a virtual memory support framework that allows gateware running on FPGAs to access the virtual memory space of its executing process in the host platform was developed. Experiment showed that, when compared with the software-centric approach, the gateware-centric implementation resulted in 1.06 to 2.49× improvement in performance while reducing the number of interrupts by half. It also eliminates all context switches to the host user process on the CPU. Apart from bus mastering direct memory access capability, little additional low-level hardware support in the COTS platform is required to implement this framework.

As such, we consider the main contribution of this work rests on the demonstration of the viability of supporting a simple gateware-centric memory access model in heterogeneous computers through the addition of a virtual memory support system in COTS FPGA. We anticipate that such a simpler memory model may eventually promote portability across platforms and productivity of application designers.

In the next section, we elaborate on the demonstration overview of the proposed system and then evaluate the performance in Section III. We make conclusions in Section IV.

II. Demonstration of the Virtual Memory System

Figure 2 shows a high-level block diagram of the proposed demo framework that provides direct virtual memory access to the user gateware. The system consists of both gateware

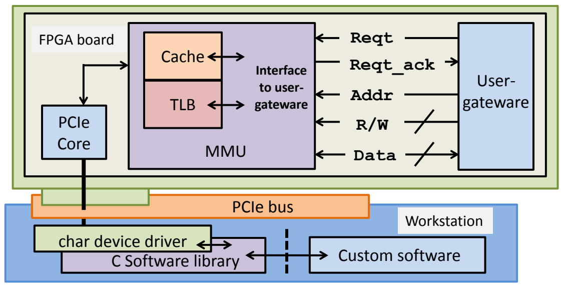

Fig. 2. System organization of the proposed framework.

components on the FPGA and software components on the CPU. On the FPGA, a memory management unit was implemented that services all memory requests from the user gateware, while on the CPU, the OS kernel was extended to coordinate with the MMU in the FPGA to ensure a consistent memory view between the CPU and the FPGA.

A. *Memory Management Unit (MMU)*

The key component of the demo framework is the MMU that provides user gateware access to the virtual memory space. It exposes a simple memory interface through which all main memory requests from the user gateware must go through. Within the MMU a TLB provides the necessary address translation to all requests. Furthermore, a memory page cache is implemented to improve memory access performance.

Translation Look-aside Buffer — The TLB provides virtual to physical address translation to the user gateware. Its action resembles that of the TLB in a conventional CPU in which a cache of the process page table is stored for rapid access. Compared to the TLB of a CPU, this TLB is relatively smaller as it serves only one user gateware process executing on the FPGA. Pages recorded in the TLB are locked and protected from other software threads by the OS driver in system memory, and are unlocked when the entries are replaced.

Our demo TLB contains 16 entries and employs LRU replacement policy. On a successful translation, the translated physical address is returned in 2 cycles. Updating of a TLB entry requires approximately 16 cycles.

Memory Page Cache — To improve speed of memory access, a small cache is also implemented alongside the TLB. In our demo implementation, this memory cache contains 4 entries, each containing a 4 kB *page* of memory. The decision to employ a page cache in the FPGA instead of the usual line cache in a CPU was mainly a consideration to improve its performance. To start, a small cache with 4 entries had made tag lookup trivial on the FPGA, greatly improving lookup time. Furthermore, we observe that instead of exhibiting random memory access patterns with occasional spatial localities as in software running on CPUs, most FPGA gateware exhibits various kinds of streaming memory access patterns. With such access pattern, it is usually favorable to prefetch a large amount of data as they are very likely to be used soon. As a result, it was decided each cache entry should hold the maximum physically contiguous segment of the memory. In this case, it was the size of a physical page. Finally, fetching a page of data from external RAM in the host also helped to amortize the bus transaction latency.

The page cache is virtually addressed so that it can be consulted in parallel to the TLB, further reducing tag lookup time. As there is only one user gateware executing on the FPGA, the choice of using virtually tagged cache has little impact on performance when compared to a physically tagged cache. Similar to the TLB, each lookup takes 2 cycles and it is employing LRU replacement policy as well.

Gateware Interface — The MMU exposes a simple access interface to the user gateware as shown in Figure 2. This interface allows the user gateware to access the virtual address space as part of the executing process. The signal pair `reqt` and `reqt_ack` indicates the completion of a memory operation. They are needed to ensure the user gateware is stalled correctly for cases such as a TLB or cache miss that results in unpredictable access latency.

PCIe Connection — To access the system main memory, our current gateware MMU implementation relies on the PCI-express (PCIe) connection of the ML505 platform that it resides on. A few technical requirements have made this the best choice for memory interfacing despite its relatively slow speed (x1 generation 1). First, the bus mastering capability of PCIe is absolutely needed to allow autonomous execution of user gateware on the FPGA without CPU's intervention. Then, in cases when the assistance of OS kernel is needed, such as updating a TLB entry, PCIe also provides a systematic interrupt mechanism from the FPGA to the CPU. Finally, it is foreseeable that the newer FPGA boards will be equipped with a faster PCIe interface due to its commonality in workstations.

B. *OS Kernel Extension*

The virtual memory system for gateware would not be complete without support from the operating system kernel running in the CPU. We extended the Linux kernel by a special device driver that serves as a kernel extension dedicated to the task of managing the TLB contents on the FPGA. In particular, the gateware MMU interrupts the CPU if an unsuccessful address translation is encountered. This interrupt triggers the execution of the interrupt handler, which in turn will look for the corresponding address translation in the kernel and update the TLB entry on the FPGA accordingly.

In the next sub-section, the interaction between the MMU on the FPGA and the OS kernel is illustrated by considering various memory access scenarios.

C. *System Workflow*

User gateware initiates an access to the virtual memory system by asserting the `Reqt` signal together with the *virtual* address in the `Addr` signal. Consequently, the MMU looks up the content from the TLB and the page cache in parallel with the supplied virtual address. Depending on the TLB and cache contents, the lookup may result in 3 different scenarios:

TLB hit, cache hit — This is the ideal case in which the requested data is already residing in the on-chip page cache. Its value is returned to the user gateware immediately for a read request. For a write transaction, the data is stored in the page cache for later write-back action.

TLB hit, cache miss — This happens due to the limited capacity of the page cache. Fortunately, since the physical address of the requested location can be computed from the on-chip TLB entry, the MMU may initiate a DMA read from the system memory directly without involving the OS. The page must have already been pinned in the main memory from previous OS action when the TLB entry was updated on the FPGA. In this case, the read latency of this request is equal to the page reading time through the PCIe bus.

In addition, if the page cache is already full, the least recently used page that is dirty must first be written back to the system memory before a new page can be loaded to the FPGA. In this case, the memory access time will be doubled.

TLB miss, cache miss — This scenario incurs the longest latency as the OS kernel must be involved to update the missing TLB entry on the FPGA.

When such a TLB miss is encountered, the MMU sends an interrupt to the CPU via the PCIe core. This initiates the execution of a predefined interrupt handler in the OS. As the handler starts, it first obtains the faulting virtual page address and checks whether the corresponding page is currently residing in the system memory. If it is not, the handler will need to page in the memory page from the backing store. Once the page is in memory, the handler will flush any corresponding CPU cache line and pin the page in memory.

The above steps are used to ensure memory coherency and consistency between the workstation and the FPGA. As soon as the above steps complete, the handler can acquire the physical page address and update the mapping to the TLB on the FPGA. In case a TLB entry needs to be evicted due to capacity limitation, the entry will be overwritten and the eviction will be reported to the OS to have the corresponding page unpinned from the system memory.

After the TLB entry is updated from the OS, the missing page cache content must also be updated with the same procedure described in the "TLB hit, cache miss" scenario.

III. Experiments & Results

To evaluate the performance of the proposed direct virtual memory methodology, micro-benchmarks on the FPGA MMU and a 2D stencil computing application were developed.

The demonstration and implementation was developed on the Xilinx Vertex 5 FPGA (XC5VLX50T) on an ML505 evaluation board. The FPGA design runs with a system clock at 62.5MHz. The workstation used is a Dell Optiplex 990 machine with Intel i5-2400 3.10GHz CPU and 4GB RAM.

A. Micro-benchmark

Two sets of micro-benchmark gateware applications were used to measure the performance of the MMU directly from the FPGA. A summary of the results is shown in Table I.

TABLE I
Summary of access latencies from gateware

Description	**Cycles**	**Time**
TLB hit	2	32 ns
TLB miss	600 to 227 000	9.6 to 3632 µs
TLB miss (page fault)	20 million	0.32 s
Cache hit	2	32 ns
Cache miss	11 100 to 18 000	177.6 to 288 µs

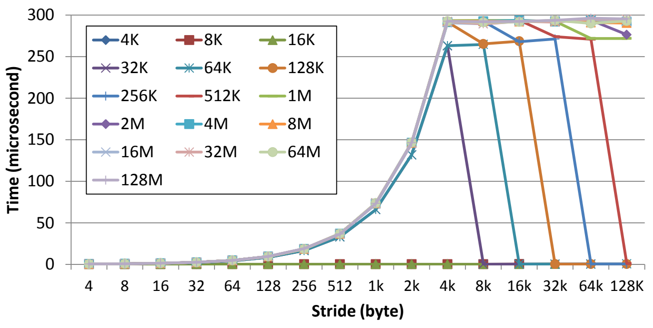

Fig. 3. Average gateware memory access time with respect to memory array and access stride sizes.

As mentioned, the MMU has a deterministic TLB/cache hit time of 2 cycles while TLB miss time varies greatly depending on the state of the OS. A gateware was developed to access cold memory addresses that generated compulsory TLB misses. The resulting memory access time was measured from the FPGA. The time to handle a TLB miss varied between 600 to 227 000 cycles with an average of around 2500 cycles.

Such long TLB miss time included 220 to 2500 cycles for the kernel to respond to interrupt requests. In addition, a large portion of time was spent on retrieving the physical address of the faulting address, pinning the page in memory and flushing the CPU cache lines involved. Finally, up to 20 million clock cycles can be spent on a TLB miss in case of a page fault in which data must be transferred from the hard disk.

To determine the page cache performance, a second gateware that generates predetermined striding memory access pattern to memory buffer of various sizes was developed.

The user gateware was written to access memory arrays of different sizes. For each array size, the gateware generated sequential read-write accesses to all array elements with various strides. The process was repeated to amortize the effect from a cold cache. These results are shown in Figure 3, which displays the familiar pattern of a single-level cache system.

For small arrays that fit entirely in the cache($\leq$ 16 kB), all memory accesses result in cache hits regardless of the stride size. As the array size grows($>$16 kB), portion of the accesses start to result in compulsory cache misses. Indeed, since the array elements are accessed sequentially, the first memory access to any new page always generates a cache miss.

Depending on the stride size, subsequent accesses will hit on the newly loaded page in the cache. In particular, the number of subsequent cache hit is always equal to $p/s - 1$ where p = page size (4 kB), s = stride size. These cache hits due to

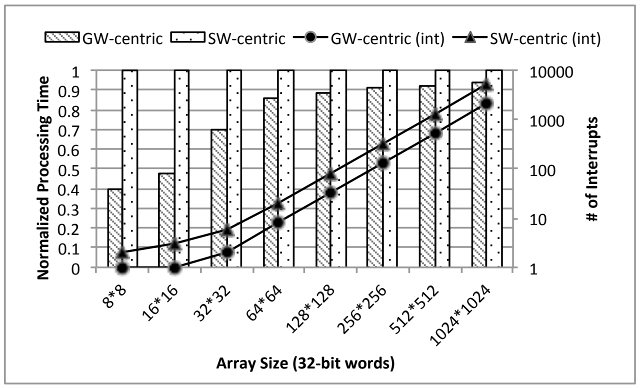

Fig. 4. Normalized 2D stencil computing time of gateware-centric implementation compared to software-centric implementation.

spatial locality result in the gradual increase in average access time relative to s as seen in the middle section of Figure 3.

The average latency continues to increase until the stride size reaches p. At this size, every data access results in a cache miss and a replacement of memory page. The average latency therefore saturates at a level equals to the cache miss time, which peaks at about 288 µs (about 18 000 cycles).

Finally, as the stride size increases further, the average latency per data access drops rapidly. It is because the stride is now so large that every access keeps falling onto the same set of 4 pages, and every access is now again a cache hit.

B. 2D Stencil Computing

To study the performance and design methodology implications of the proposed memory access model in real-life situations, a 2D stencil computing application was developed and will be used as the demo application.

In our benchmark application, the input is an $n \times n$ 2D array (a0). Each element is updated depending on values of its 8 immediate neighbors and itself, forming a 3×3 update window. An additional array (a1) the same size as a0 is allocated to hold intermediate values for computation. Two versions of this application were implemented with both gateware-centric and software-centric approach. In both versions, a0 and a1 are stored in the system main memory.

Performance results of GW-CENTRIC are normalized to that of SW-CENTRIC with respect to the left axis in Figure 4. Also displayed in the figure is the number of interrupts generated for each iteration of stencil computing. They correspond to the number of TLB misses in GW-CENTRIC and the number of DMA data transfer between CPU and FPGA in SW-CENTRIC.

As shown in Figure 4, the GW-CENTRIC implementation consistently outperforms the SW-CENTRIC implementation especially when n is small. As the input 2D array is cached as 4 kB memory pages on the FPGA, accesses to the elements in a0 benefit from such spatial locality and hit on the cache most of the time. Moreover, the same copy of data must be resent to the FPGA multiple times as the window slides across a0. The limited I/O bandwidth on the FPGA add-on card significantly affects the performance of such redundant data transfer.

TABLE II
RESOURCE CONSUMPTION OF THE PROPOSED SYSTEM. PERCENTAGE VALUES ARE RELATIVE TO AVAILABLE RESOURCES ON THE TARGET FPGA

Modules	**Flip-Flop**		**LUT**		**BRAM**	
TLB	1620	5 %	2008	6 %	36 kB	1 %
Cache	606	2 %	1160	4 %	144 kB	6 %
MMU	2841	9 %	3947	13 %	198 kB	9 %
Overall	6202	21 %	7078	24 %	414 kB	19 %

As n increases, however, the performance gap between the two shrinks. Due to the much increased array size and the fact that stencil processing is inherently stream-oriented, new pages of data from a0 must be loaded to the FPGA in both implementations. Because of the large gap between cache hit and TLB miss time, the performance advantage of GW-CENTRIC begins to shrink.

In terms of interrupt, GW-CENTRIC consistently generates less than half the number of interrupts on the CPU than SW-CENTRIC. Furthermore, interrupts in SW-CENTRIC correspond to data I/O and require waking up the host software process to handle. It not only causes significant additional load to the CPU, but also delays the overall stencil computation. With reduced load on CPU and higher performance, the advantage of GW-CENTRIC approach is clearly demonstrated in this case.

C. Resource Consumption

Table II summarizes the resource consumptions of the proposed system. On the FPGA chosen, the PCIe core together with the MMU consumes less than a quarter of resources. In particular, the MMU consumes less than 10 % of flip-flops and BRAM and 13 % of LUTs. This includes the TLB, the page cache and their control logic. We consider such moderate consumption reasonable given the functionality they provide.

IV. CONCLUSIONS

This demo presented a virtual memory system framework for user gateware that reduces the number of context switches between FPGA and CPU by half with reasonable performance improvement. With such capability, application designers and compilers may focus on optimizing the application for the specific compute platform instead of devoting efforts in handling the intricate data movements between CPU and FPGA. In the future, we intend to extend the system to optimize for FPGA specific application requirements such as concurrent memory access and improve the cache system to accommodate the diverse granularity requirements in FPGA applications.

REFERENCES

[1] H. K.-H. So and R. Brodersen, “A Unified Hardware/Software Runtime Environment for FPGA-Based Reconfigurable Computers using BORPH,” *Trans. on Embedded Comp. Sys.*, vol. 7, no. 2, pp. 1–28, 2008.
[2] A. Ismail and L. Shannon, “Fuse: Front-end user framework for o/s abstraction of hardware accelerators,” in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2011, pp. 170–177.
[3] T. Brewer, “Instruction Set Innovations for the Convey HC-1 Computer,” *Micro, IEEE*, vol. 30, no. 2, pp. 70 –79, march-april 2010.