TACC Technical Report TR-14-10

Low Level Microbenchmarks of Processor to FPGA Memory-Mapped IO

John D. McCalpin
Texas Advanced Computing Center,
The University of Texas at Austin
mccalpin@tacc.utexas.edu

February 21, 2014

This technical report may be an early version of a document intended for publication in a journal or proceedings. Since changes may be made before publication, readers desiring to cite this work should first check to see if a newer, published version exists in a journal or conference proceedings.

This work was funded in part by the National Science Foundation under grants CCF-1240652 and OCI-1134872.

Abstract

The Input/Output (IO) capabilities of modern microprocessor-based systems are not ideally suited to the needs of tightly coupled accelerators, yet IO-attachment remains the most practical choice for assembling heterogeneous systems. This report describes the range of capabilities of a recent high-performance processor (the Intel Xeon E3-1270) with regard to using memory-mapped IO over PCI Express as the interface to an external accelerator (an Altera Stratix IV FPGA), and presents a model for configuring a system to optimize fine-grained communication and synchronization operations. These theoretical analyses are augmented by results of low-level measurements of load and store performance for processor-initiated operations to registers on the FPGA and to SRAM memory attached to the FPGA.

1 Introduction

The Input/Output (IO) architecture in microprocessor-based systems has evolved only slowly from its early roots in supporting disks and graphics displays, and currently falls far short of both the performance limits dictated by physics and the performance required to exploit tightly-coupled accelerators.

This report considers the performance attributes of low-level interactions between a host processor and an external accelerator attached to a PCI Express device, with special attention to the very short data transfers required for synchronization and status checking operations. While DMA-based transfer is clearly the most efficient approach for large data transfers, the short data transfers common to synchronization and polling are more effectively handled by direct processor access to registers and/or data areas on the external device. Following standard practice, we assume that the external accelerator exposes both its internal status registers and its external QDR SRAM array as memory-mapped IO (MMIO) devices on the host.

Departing from standard practice, this report considers the functionality and performance implications of all of the "memory types" supported by the x86 architecture. Although only a few of these types were tested directly, low-level microbenchmarking tests provide latency, concurrency, overhead, and net transfer rate values that are used with analytical models to predict the performance of modes of operation that could not be tested directly during the evaluation period.

2 Background: Communication and Architecture

It is important to note that "communication" does not exist as an explicit feature in the architectural specification documents defining the major computer architectures in use today. The term does occur a handful of times in the AMD64 and Intel64 architectural specifications, but only as an aside, never as an explicit architectural concept. E.g., the Intel SW developers guide [2] mentions that a locked (atomic) read-modify-write operation can be used as a mechanism "to allow reliable communications between processors in multiprocessor systems".

The difference between "allowing" communication and architecturally supporting communication is critical. Current architectures "allow" communication to be implemented indirectly using sequences of (partially) ordered loads and stores to a shared memory space. Requiring multiple steps to implement communication limits performance – an effect which is exacerbated by both the use of transparent (uncontrollable) caches and the requirement that memory references be free of side effects.

Since “communication” is not an architectural feature, it is not possible to build optimized hardware to support it. Similarly, since the memory references that lead (indirectly) to ”communication” are not distinguishable from “local” memory accesses, there is no way for the hardware to process those references using (different) protocol choices that are more efficient than the default protocols (which are optimized for single-thread data reuse).

The situation is similar for “communication” between processors and IO devices, as discussed in the next section.

3 Input/Output In Recent Microprocessors

In order to obtain high throughput, most IO devices contain a “DMA engine” capable of transferring blocks of data between the IO device and system memory (or vice-versa). Once the block copy is complete, the IO device either sends an interrupt to the host OS or updates a “flag” variable that the processor can poll. This approach works extremely well for large transfers, but suffers from excessive overhead when the payload to be transferred is small.

Modern microprocessor-based systems also support “Memory Mapped IO” (MMIO) – mapping the registers and/or memory of IO devices to physical addresses in the host processor’s address space. This allows standard load and store instructions to be used to obtain information from the IO device or to control its data and/or function (including programming the DMA engine(s) on the IO device).

Unfortunately, the performance associated with most uses of MMIO is extremely poor. System design appears to assume that the majority of the IO will be associated with large transfers, while the mechanisms associated with short transfers remain unoptimized. This is fine for traditional uses such as disk IO, but is a serious obstacle to effective use of external accelerators for tightly-coupled computations (requiring fine-grained synchronization).

For tightly coupled accelerators, the overhead of programming an external DMA engine and handling completion interrupts can be unacceptably high. In such cases it may be preferable to allow the host processor to directly handle the IO operations, and it is this situation that we consider here. This mode of operation is often referred to as “Programmed Input/Output” (PIO) in reference to the data transfer mode used by early IDE disk drives in PCs. To avoid confusion with the specifics of legacy IDE interfaces, I will refer to this mode of operation simply as “MMIO” or will describe it as direct processor access to accelerator control functions and/or data space.

3.1 Memory Types

The x86 processor architecture supports several different "Memory Types" controlled by a combination of "Memory Type Range Registers" (which control behavior for physical address ranges) and "Page Attribute Table" entries (which control behavior at page-level granularity). Although the details of how these are combined can be complex, the result is the ability to define five different "memory types", each of which provides a different combination of cacheability, speculation, and memory ordering models.

The "memory types" are:

Uncached (UC) Neither reads nor writes are cached. No speculation is allowed. All UC accesses are fully serialized with respect to all other memory references.

Write Combining (WC) Reads are not cached, but speculative reads are allowed. Writes are "combined" in "write-combining buffers" to allow high-throughput. WC writes are not strongly ordered with respect to cached writes (below) or other WC writes.

Write Through (WT) Reads are cached, and speculative reads are allowed. Writes go "through" the caches to memory (updating any copies of the line that are present in the caches), but are not combined into full-cache-line transfers. WT writes obey the normal (strong) processor ordering model.

Write Protect (WP) The same as Write Through (WT) except that stores invalidate any copies of the line in the caches rather than updating them.

Write Back (WB) This the the "normal" cached mode of operation. Reads and writes are cached and follow the x86 strong processor ordering model. Reads can be issued speculatively.

3.2 Memory Type for High Performance IO

Given the existing hardware capabilities, what memory types are appropriate for MMIO?

First we have to split off one special case – IO devices for which MMIO accesses have side effects. For this case only the UC memory type can be allowed because only the UC memory type prohibits speculation. An example of an access with a side effect is popping a value off of a hardware-based stack located in MMIO space. If the MMIO space is mapped with a type that allows speculation, a load from the hardware stack address may be executed speculatively before a preceding conditional branch has been resolved. If the branch resolves to the path away from the load, the speculatively loaded value will be discarded and that value from the stack will be lost.

(Aside: I have worked with processor design engineers at many companies on this issue and have found no case in which the engineers were willing to guarantee that any

approach to avoid speculation other than the use of UC would work reliably.)

Uncached accesses are, unfortunately, terrible for performance. To achieve strong ordering and a lack of speculation, when an uncached memory access appears in the instruction stream, processors typically perform (something similar to) the following steps:

1. Wait until all prior instructions with memory references have retired.
2. Wait until all prior speculatively executed instructions have been resolved and either retired or rejected.
3. Execute the uncached memory access, reading or writing only the specific bytes requested by the instruction.
4. Wait until the uncached memory access is completely finished and acknowledged by the target.
5. Begin executing the next instruction in the instruction stream.

Uncached memory references only support 1 Byte to 8 Byte arguments referencing general-purpose registers, so you lose the ability to transfer 64 Byte cache-line blocks. This costs at least a factor of 8 in performance. Uncached memory references allow no concurrency, so the bandwidth is the number of bytes requested divided by the latency. This costs at least another factor of 8 in performance. Uncached memory references allow no prefetching. This costs another factor of 2 in performance. MMIO accesses to an FPGA device typically have much higher latency than accesses to system memory. The experiments below will provide details, but for the system under test this reduces bandwidth by almost another factor of 8. Combining these four factors suggests that bandwidth for uncached loads will be (at most) approximately 1/1000 the bandwidth for cached loads, and measurements on the system under test support this ratio.

So are there other options for memory mapping for MMIO?

Yes. As long as we are dealing with MMIO regions for which there are no side effects, the WC type can always be used and the WP and WT types may be usable (subject to restrictions and limitations to be discussed below). Only the WB type is absolutely prohibited for MMIO, due to lack of support for several of the required coherence transaction types[1] [2].

The WC type was designed to allow processors to write to graphics frame buffers at high rates. High rates require both block transfers and support of concurrency, so WC allows both. The block size used by WC matches the cache line size (64 Bytes)

1. See "Notes on Cached Access to Memory-Mapped IO Regions" http://sites.utexas.edu/jdm4372/2013/05/29/
2. One could imagine a set of transaction re-mappings for the WB type that would allow restricted use of WB for MMIO regions, but these are definitely not supported by any Intel or AMD processors of the last decade.

on Intel and AMD processors, while the number of lines that can be concurrently ”in flight” is implementation-dependent – typical values are between 4 and 10. Aside: Intel and AMD processors support “non-temporal” stores to WB memory using special non-temporal store instructions. These allow stores to bypass the cache and therefore allow the hardware to eliminate the usual read of the line into the cache before it is over- written. These non-temporal store instructions are not required for WC memory spaces (since all stores go through the write-combining buffers), but they are permitted (and perhaps preferred since they emphasize that the stores in question follow WC semantics).

The WC type does not allow reads to be cached, and therefore does not allow for 64 Byte block loads from MMIO space (with one exception to be described below). WC does allow speculative reads, which sometimes allow concurrent reads to MMIO space, but I have been unable to exploit this reliably. Fortunately, both the WP and WT memory types allow cached reads and speculation, so these can provide block transfers and may provide concurrent transfers as well (depending on implementation details).

I was unable to test the WP or WT memory types during this evaluation. To set up a region as WP or WT, one needs to both set up an MTRR marking this address range as WP or WT (this part was easy using the Linux /proc/mtrr interface) and convince the kernel to set up the PAT bits as WP, WT, or WB. This latter operation was the stumbling block. The Linux kernel (up to version 3.16) is aware of the WP and WT PAT types, but does not provide any support to generate these types. The kernel actually goes out of its way to make these types more difficult to use, by defining all of the entries in the PAT table to correspond to UC, WC, or WB types. The combination of a WP or WT MTRR type with a WB PAT type does produce the desired WP or WT mapping, but the Linux kernel ignores the user request in mapping MMIO regions and maps them all UC or WC – even when the device driver code explicitly calls the Linux kernel function “ioremap_cache”. I was unable to understand the Linux kernel coding style and flow of control well enough to override this functionality during the evaluation period.

As mentioned above, there is one exception that allows block reads from WC memory – the MOVNTDQA instruction. This allows (but does not require) that an implementation load a 64 Byte block from the WC region and serve up to four 16 Byte (aligned) MOVNTDQA instructions from that buffer. This instruction was added by Intel with version 4.1 of the SSE extensions for processors launched in 2008 or later. Unfortunately I was working at AMD at the time and was unaware of this Intel extension (which AMD did not introduce until late 2011, three years after I left the company). I discovered this instruction a few weeks after the evaluation period for the test system, so I was not able to test it. The Intel documentation on the instruction allows for

significant variation in the implementation, but the initial disclosure of the instruction (SSE Streaming Loads)[3] mentions a 7.5x speedup relative to the standard MOVDQA instruction. Since there are 4 16 Byte loads per 64 Byte buffer, this suggests that the mechanism supports an effective concurrency of 2 64 Byte reads. While this regains only a small fraction of the performance loss suffered by uncached loads, it should still be quite useful to extend the domain over which direct processor MMIO is the most efficient mechanism for interfacing with a tightly-coupled coprocessor. This is investigated using an analytical performance model in a later section.

More discussion on these topics is available at:

- Notes on Cached Access to Memory-Mapped IO Regions
  http://sites.utexas.edu/jdm4372/2013/05/29/
- Coherence with Cached Memory-Mapped IO
  http://sites.utexas.edu/jdm4372/2013/05/30/
- Intel Developer Forum Topic: Cache Write Back for PCI Devices
  https://software.intel.com/en-us/forums/topic/393070

4 Microbenchmarks

4.1 Methodology

The system under test employed a Xeon E3-1270 ("Sandy Bridge") processor in a SuperMicro server system. The nominal frequency of the processor cores is 3.4 GHz, but Turbo mode allows a single core to operate at up to 3.8 GHz. To reduce confusion and performance variability, Turbo mode was disabled and all cores were set to a fixed 3.4 GHz operating frequency. An Altera Stratix IV FPGA was connected via an 8-bit PCI Express (generation 2, 5.0 Gbs) interface to the Altera "hard IP" PCIe controller block. The FPGA was also configured with external QDR II SRAM memory for bulk random access storage with 64-bit access granularity.

The FPGA was programmed to provide additional functionality (not directly relevant here), which provided a base configuration on which to perform simple experiments with load and store latency and bandwidth. This base configuration resulted in a 200 MHz (5 ns) FPGA clock frequency.

The system was running a RHEL 6.2 Linux distribution (2.6.32-220 kernel) with a custom device driver to provide access to the internal configuration and status registers of the FPGA as well as to the external SRAM attached to the FPGA. The device driver

3. The full URL is https://software.intel.com/en-us/articles/increasing-memory-throughput-with-intel-streaming-simd-extensions-4-intel-sse4-streaming-load

provided both `ioctl` interfaces to read and write to the MMIO address space of the FPGA and an `mmap` interface to provide a user-mode pointer for direct access to the MMIO address space. The `mmap` functionality was used to test performance with both UC and WC memory types.

4.2 Results

4.2.1 Device Driver Overhead

The first overhead to consider in IO performance is the time required to transition to and from kernel mode when performing MMIO operations through a Linux kernel device driver. Several of the microbenchmarks were used to estimate device driver overhead, delivering reasonably consistent results for the cost of using the `ioctl`-based interface through the device driver.

The first approach used to estimate device driver overhead was a write bandwidth test. This test code uses an `ioctl` call to request that the device driver read a block of data from user space and copy it to a user-specified range of addresses in the off-chip QDR SRAM attached to the FPGA.

To estimate the driver overhead, I set up a series of cases that copied 1 MiB of data from the processor to the FPGA using progressively smaller transfer sizes (and progressively larger numbers of calls into the drive to copy the sub-blocks of the 1 MiB range). The results were modeled based on the assumption that the actual data transfer time was fixed (since all cases moved exactly 1 MiB) and that the overhead of the kernel calls was linearly proportional to the number of calls. Least-squares curve fitting showed very good agreement between the model and data with a device driver overhead of 490 cycles per call (144 ns at 3.4 GHz).

Several additional test cases performing read operations were also used to test the overhead of the device driver calls. A direct comparison of the latency to read 8 bytes through the device driver and using a user-space pointer gave an overhead of 411 cycles (121 ns). Increasing the read size to 64 Bytes provided an estimated device driver overhead of 405 cycles (119 ns). An indirect estimate provided by "spinning" on a flag and then reading a different value when the flag became "ready" suggested a device driver overhead of 487 cycles (143 ns).

This range of overhead values was considered unacceptably high for fine-grained synchronization, so further testing emphasized direct load/store access to MMIO space using a pointer provided by the `mmap` function of the device driver.

4.3 User Space Read Latency

The analyses used to estimate device driver overhead also provided either estimates or direct measurements of the raw hardware read latency. To minimize the impact of cache misses, a very small block of data (64 Bytes) was loaded from the FPGA into the processor eight times and the elapsed time was measured using a very low-overhead inline RDTSCP instruction.

In these cases the MMIO region was mapped to the processor using a WC memory type. Reads to WC regions are uncached, but speculation is allowed. During these experiments there were several instances of unexpectedly low latency measurements that might have been indications of read concurrency – presumably due to overlap between (for example) a status register read and the speculative execution of the actual data read after a compare and branch based on the status. Unfortunately these occurrences, while consistent for a given compilation of the device driver and test code, were not persistent across the changes to code intended to investigate the anomalies. The results presented and discussed here exclude all cases that exhibited anomalous timings. This was enforced by adding either an `fence` or `mfence` instruction between reads.

Direct measurement of user-space status register read latency showed an average of 1500 cycles (441 ns) per 64-bit load. The inline user-space time stamp counter reads contributed less than 10 cycles (2.5 ns) to this value.

A different case that mixed reads from a (different) status register and reads from the off-chip SRAM memory provided estimates of 1475 cycles (433 ns) for the status register read and 1725 cycles (507 ns) for the off-chip SRAM reads. The small difference in read latency for the two status registers might reflect an actual latency difference (due to the different locations of the status register within the FPGA) or might reflect measurement and/or modeling errors. The much larger difference between the status register read latency and the SRAM read latency is almost certainly real, and is expected to be primarily attributable to the need to traverse the SRAM control logic. (The raw SRAM access latency of 7.5 ns contributes only about 10% of the overall 74 ns latency increase.)

4.4 Write Bandwidth

The PCI Express interface (8 lanes, generation 2, 5.0 Gbs) has a peak bit rate of 40 Gbs per direction, but the 8/10 encoding reduces the raw data rate to 32 Gbs (4 GB/s) per direction. Protocol overheads vary with maximum packet length and transaction type, but are typically in the range of 20-24 Bytes for a 64 Byte “posted write” transfer, leading to best-case payload efficiencies of about 72%-76%.

Measured values for memory-to-FPGA transfers attained maximum values of about 2.92 GB/s, or 73% of the peak rate, which is completely consistent with the expected overheads[4]. For this bandwidth measurement, the time required to load the data from memory (before writing it to the FPGA) was minimal, given the sustained read rates of over 16 GB/s and the ability of the hardware prefetchers to sustain those reads in parallel with the WC stores. Note that the device driver performs the copy from memory to FPGA one cache line at a time, reading a line from memory, then writing that data to MMIO – repeating until all cache lines in the region have been copied. This allows nearly perfect overlap of the read traffic and the MMIO write traffic.

5 Discussion

5.1 Modeling the Performance Impact of Streaming Loads

Based on Intel's initial disclosure of the MOVNTDQA instruction, we assume that the implementations support at least 2 concurrent 64 Byte load transactions from WC-mapped MMIO space. Intel recommends using tightly packed instructions to load all data from each aligned 64-Byte block as quickly as possible to minimize the likelihood that the buffer will be emptied and will need to be refilled (e.g., if an interrupt occurs in the middle of the sequence of reads). The reported speedup of 7.5x relative to using 8 Byte uncached loads suggests that the mechanism will provide a nearly linear reduction in latency for block sizes between 16 Bytes and 128 Bytes (relative to using only 8 Byte uncached loads).

More information and recommendations on usage and limitations of the MOVNTDQA instruction are in [1, §12.10.3], including an example of how to use this instruction with a processor as a "consumer" and a PCI device as a "producer".

A simple model was developed to estimate the minimum time required for a processor to obtain data from a memory-mapped accelerator using either a DMA engine (followed by polling) vs direct processor-based access. Although performance for very short payloads was poor in both cases, the additional interactions required to program the DMA engine and to poll the DMA engine completion flag make the direct processor-based access more efficient for short messages. The DMA-based approach remains more efficient for long messages due to the increased concurrency available (relative to the assumed streaming loads implementation).

For the model parameters chosen, the direct processor-based approach is estimated to incur about 1/2 the latency of an optimized DMA implementation for short messages

4. The slightly higher efficiencies commonly seen with DMA transactions exploit larger payload sizes to reduce the relative overhead of the protocol.

(less than 128 Bytes), with smaller advantages for transfer sizes between 128 Bytes and 512 Bytes. Transfers of 1 KiB and larger are expected to be faster using DMA (due to the limited concurrency available via the streaming load mechanism).

The only ways to significantly improve these values are to:

1. Reduce the latency of reads and writes to the FPGA, or
2. Change the protocol to allow block reads with side effects (to reduce the number of latencies required in the critical path).

5.2 Practical Latency Limits

Complaining that performance is "too low" is not useful unless it can be demonstrated that existing performance limits are due to architecture choices and not dictated by physics. In the case of tightly-coupled accelerators, we can point to existing implementations that are capable of performing some tasks very rapidly, while other tasks (with similar physical constraints) perform very slowly. An example of high performance provided by AMD's Coherent HyperTransport interface and by Intel's QPI interconnect. Both of these provide a cache-coherent interconnect fabric across multiple processor chips, and both demonstrate that a core can probe its caches, send a snoop request to the other chip, have the other chip probe its caches, and return a snoop response to the originating core – all in about 60 - 70 ns. In some cases data can be returned almost as fast, but many special cases must be considered – some of which are substantially slower than the best cases. The best case I have measured on a 2-socket Xeon E5-2680 ("Sandy Bridge EP") system is 67 ns for local memory access and 71 ns for remote cache-to-cache intervention (when running at 3.1 GHz). This cache-to-cache intervention latency is a factor of 7.5 faster than reading data from the off-chip SRAM on the FPGA.

Of this round trip time, some portion is required to actually probe the cache tags on each chip. It is difficult to obtain precise values for this part of the time budget, but values of 10 ns to probe all of the local caches before sending the probe request to the other chip and 10 ns to probe all of the remote caches before the probe response can be returned are plausible. This leaves 40 - 45 ns as the round trip time for the remainder of the processing, or about 20 ns as a plausible "demonstrated latency" for one-way transmission of a "message" from one chip to another.

The one-way message latency has important implications for bandwidth and required concurrency, which are related by

$$\text{Latency} = \text{Concurrency} \times \text{Bandwidth}$$

This shows, for example, that the concurrency required to sustain a particular level of bandwidth increases linearly with increases in latency. As a quantitative example,

an interface capable of 6.4 GB/s of unidirectional bandwidth (e.g., PCI express, 8 bit, generation 3 after protocol overhead) with a latency of 60 ns would require a concurrency of 384 Bytes, or 6 64-bit cache lines in flight at all times – easily managed by a single core. Increasing the latency to 420 ns would increase the required concurrency to 42 cache lines, which is beyond the capability of a single processor core in current microprocessor implementations.

An alternate description of the "cost" of synchronization is to relate the overhead to the amount of work that could have been done by the accelerator during the synchronization interval. Assuming 1/2 of a read latency to "push" a short data item to the accelerator, 1.5 read latencies (average) to poll the completion flag, and 1 read latency to obtain the data, we have a minimum execution time of approximately 3 read latencies (1200 ns on the reference system) even if the accelerator is able to perform the requested operation at infinite speed. During this 1200 ns period, slightly over 4000 processor cycles would have elapsed, corresponding to a theoretical maximum of 16000 instructions or over 32000 64-bit floating-point operations for a single core. Assuming a 1 TFLOPS peak floating point rate on the accelerator, this elapsed synchronization time also corresponds to 1.2 million floating point operations on the accelerator.

6 Summary

Modern microprocessor-based systems have IO infrastructures that are not well suited to the requirements of tightly-coupled accelerators. This report presents measurements of read and write performance between a processor and an FPGA, using both device driver and user-space interfaces, and discusses alternate configurations that have the potential to reduce overhead and latency. Although some improvements to performance can be obtained (e.g., by reducing latency) the primary problem is the excessive number of low-level transactions required to "hand off" data in each direction and the inability to effectively pipeline these transactions to MMIO space. It is clear that architectural changes in both the protocols and the processors will be required to enable the development of accelerators that can be effective in applications requiring fine-grained communication and synchronization.

References

[1] Intel Corporation. *Intel64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*, 2014. Document 253665-051.

[2] Intel Corporation. *Intel64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*, 2014. Document 325384-051.