FARM: A Prototyping Environment for Tightly-Coupled, Heterogeneous Architectures

Tayo Oguntebi, Sungpack Hong, Jared Casper, Nathan Bronson, Christos Kozyrakis, Kunle Olukotun
Stanford University
{*tayo, hongsup, jaredc, nbronson, kozyraki, kunle*}*@stanford.edu*

***Abstract*—Computer architectures are increasingly turning to parallelism and heterogeneity as solutions for boosting performance in the face of power constraints. As this trend continues, the challenges of simulating and evaluating these architectures have grown. Hardware prototypes provide deeper insight into these systems when compared to simulators, but are traditionally more difficult and costly to build.**

We present the Flexible Architecture Research Machine (FARM), a hardware prototyping system based on an FPGA coherently connected to a multiprocessor system. FARM substantially reduces the difficulty and cost of building hardware prototypes by providing a ready-made framework for communicating with a custom design on the FPGA. FARM ensures efficient, low-latency communication with the FPGA via a variety of mechanisms, allowing a wide range of applications to effectively utilize the system. FARM's coherent FPGA includes a cache and participates in coherence activities with the processors. This tight coupling allows for realistic, innovative architecture prototypes that would otherwise be extremely difficult to simulate. We evaluate FARM by providing the reader with a profile of the overheads introduced across the full range of communication mechanisms. This will guide the potential FARM user towards an optimal configuration when designing his prototype.

***Keywords*-prototyping; coherent FPGA; FPGA communication; HyperTransport; accelerators; coprocessors**

I. Introduction

Computer architecture researchers are aggressively exploring new avenues for delivering performance in future systems. Many of these architectures, such as the heterogeneous ones, are fundamentally different from existing hardware and difficult to accurately model using traditional simulators. Cell phone chips, encryption engines, CPU-GPU hybrids, and on-chip NICs are all examples of special functions that are tightly coupled with CPUs. New hardware prototypes are therefore extremely useful, being faster and more accurate than simulators. In addition to providing better insight into the system and being able to run larger and more realistic pieces of code (such as an OS), prototyping allows researchers to find bugs and design holes earlier in the development cycle.

In this paper, we present a unique prototyping environment called the Flexible Architecture Research Machine (FARM). FARM is based on an FPGA that is coherently tied to a multiprocessor system. Effectively, this means that the FPGA contains a cache and participates in coherence activities with the processors via the system's coherence protocol. Throughout this paper we refer to an FPGA connected coherently as a "coherent FPGA." Coherent FPGAs allow for prototyping of some interesting segments of the architectural design space. For example, architectures requiring rapid, fine-grained communication between different elements can be easily represented using FARM. Ideas involving modifications to memory traffic, coherence protocols, and related pursuits can also be implemented and observed at the hardware level, since the FPGA is part of the coherence fabric (Section V). The close coupling also obviates the need for soft cores or other processors on the FPGA in many cases, since general computation can be done on the (nearby) processors. Section II provides details about the system architecture and implementation of FARM. In addition to prototyping, FARM's architecture is naturally well-suited to exploring *accelerated architectures*, with the FPGA functioning as the accelerator (or coprocessor).

Using a tightly-coupled coherent FPGA, whether as an accelerator or for prototyping, presents communication and sharing challenges. One must provide efficient and low-latency methods of communication to and from the FPGA. When functioning in the capacity of an accelerator, in particular, it is very necessary to understand the behavior of the *communication mechanisms* offered by FARM. The mechanisms include: traditional memory-mapped registers (MMRs), a streaming interface, and a coherent cached interface. Section III details these methods of communication and suggests how one important application characteristic, frequency of synchronization, could affect the choice of communication mechanism.

System designers must understand the tradeoffs and overheads that accompany each communication type when using it to accelerate applications with various characteristics, especially differing levels of synchronization between the FPGA and the processors. In particular, knowledge of the execution overhead introduced by using a dedicated remote accelerator would suggest a minimum for the speedup benefits gained when using that accelerator. Furthermore, this overhead is not constant, but rather a function of the type of communication chosen as well as other characteristics, such as latency and synchronization. Section IV explores these issues by presenting the performance of a synthetic benchmark on FARM for all communication mechanisms and various other factors. Such data should influence users of FARM-like systems when deciding on implementations of heterogeneous prototypes or coprocessors.

A. Related Work

The FARM prototyping environment follows in the tradition of previous FPGA-based hardware emulation systems such as the Rapid Prototyping engine for Multiprocessors (RPM) [1]. RPM focused on prototyping multiprocessor architectures where FPGAs are used primarily for gluing together symmetric cores, but not much for computation. RAMP White [2] is a similar approach, prototyping an entire SMP system with an FPGA, including CPU cores and a coherency controller. We differ in that our approach is more directed at evaluating heterogeneous architectures, where the FPGA prototypes a special-purpose module (e.g. an energy-efficient accelerator) attached to high-performance CPUs. Convey Computer Corporation's HC-1 is a high-performance computing node that features a coprocessor with multiple FPGAs and a coherent cache [3]. Convey's machines are different in that they optimize for memory bandwidth in high-performance, data-parallel applications. The coprocessor's cache is usually only used for things like synchronizing the start sequence. Recently, AMD researchers have also implemented a coherent FPGA [4]. AMD's and our system use different versions of the University of Heidelberg's cHT core to handle link-level details of the protocol[1], but AMD does not give a thorough analysis of system overheads for various configurations and usages.

Indeed, there has not been much discussion on how these coherent FPGA systems can be well-utilized, and what kinds of applications can benefit from them. In this paper we discuss issues such as system utilization and present some key considerations to account for when building with these systems. We also provide the detailed design and implementation of our system.

The major contributions of this paper are:

- We present FARM, a novel prototyping environment based on the use of a coherent FPGA. We detail its design, implementation, and characteristics.
- We describe useful mechanisms for processor-FPGA communication and evaluate these mechanisms on FARM with varying application characteristics.

II. FARM

This section presents the design details of FARM. We begin with a description of the system architecture and the hardware specifications of our particular implementation. We then describe the usage of the FPGA in FARM and detail the design and structure of some of our key units. We also reveal our implementation of the coherent HyperTransport protocol layer and describe methods and strategies for efficiently communicating coherently with CPUs.

[1]The cHT core was provided by the University of Heidelberg [5] under an AMD NDA. We made modifications and extensions to the core to improve functionality, increase performance and integrate with the FARM platform.

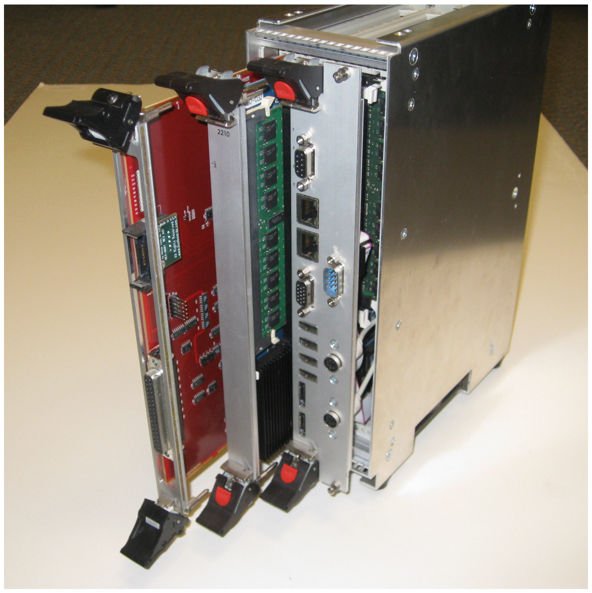


Figure 2. Photo of the Procyon system with a main board, CPU board, and FPGA board.

A. FARM System Architecture

FARM is implemented as an FPGA coherently connected to two commodity CPUs. The three chips are logically connected using point-to-point coherent HyperTransport (HT) links. Figure 1 shows a diagram of the system topology, along with bandwidth and latency measurements, as well as the high level design of the FARM hardware. Memory is attached to each CPU node (not shown). Latency measurements in the figure represent one-way trip time for a packet from transmission to reception, including de-serialization and buffering logic.

We used the Procyon system, developed by A&D Technology Inc. [6], as a baseline in the construction of the FARM prototype. Procyon is organized as a set of three daughter boards inter-connected by a common backplane via HyperTransport. Figure 2 shows a photograph of the Procyon system. The first board is a full system board featuring an AMD Opteron CPU, some memory, and standard system interfaces such as USB and GigE NIC. The second board houses another Opteron CPU and additional memory. The third board is an FPGA board with an Altera Stratix II EP2S130 and support components used for programming and debugging the FPGA. The photograph shows the FPGA board, secondary CPU board, and full system board from left to right, respectively. Table I gives a detailed listing of FARM's hardware specifications.

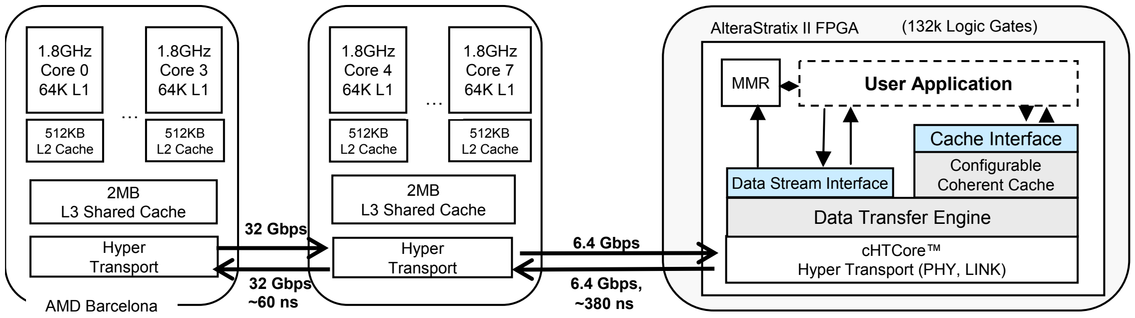


Figure 1. Diagram of the Procyon system with the FARM hardware on the FPGA.

CPU Type	AMD *Barcelona* 4-core (2 CPUs)	DRAM	3GB (2GB on main system board)
Clock Freq	1.8 GHz	HT Link Type	HyperTransport: 16-bit links
L1 Cache	Private: 64KB Data, 64KB Instr	CPU-CPU HT Freq	HT1000 (1000 MT/s)
L2 Cache	Private: 512KB Unified	CPU-FPGA HT Freq	HT400 (400 MT/s)
L3 Cache	Shared: 2MB	FPGA Device	Stratix II EP2S130
Physical Topology	3 boards connected via backplane	Logical Topology	Single chain of point-to-point links

Table I
HARDWARE SPECIFICATIONS OF THE FARM SYSTEM.

Our FARM device driver is somewhat unique in that it is the driver for a coherent device, which looks quite different to the OS than a normal non-coherent device. To allow for flexibility in communication with the FPGA, the driver reconfigures the system's DRAM address map (in the MTRRs and PCI configuration space) to map a section of the physical address space above actual physical memory to "DRAM" on the FPGA. We must keep this memory hidden from the OS to prevent it from being used for normal purposes. Using the mmap mechanism, these addresses are mapped directly into the user program's virtual address space. The FPGA then acts as the memory controller for this address space, allowing the user program to read and write directly to the FPGA. This memory region can be marked as uncacheable, write-combining, write-through, or write-back. Our original design marked this "FARM memory" as uncacheable to allow for communication with FARM that bypassed the cache. However, the *Barcelona* CPUs impose very strict consistency guarantees on uncacheable memory, so we instead mark this section as write-combining in FARM (see Section III). The FARM device driver is also used to pin memory pages and return their physical address in order to facilitate coherent communication from the FPGA to the processor. An alternative, albeit more complicated, solution would be to maintain a coherent TLB on the FPGA.

Reconfigurability in a prototype built with FARM is provided via the attached FPGA. The FPGA houses modules that allow for general coherent connectivity to the processors as well as a means by which the coprocessor or accelerator can use these modules. As shown in Figure 1, the FARM platform implements a version of AMD's proprietary coherence protocol, called coherent HyperTransport (cHT). With some exceptions, the cHT definition is a superset of HyperTransport that allows for the interconnection of CPUs, memory controllers, and other coherent actors. Coherent HyperTransport implements a MOESI coherence protocol. The cHT core, also described in the introduction, handles only link-level details of the protocol such as flow control, CRC generation, CRC checking, and link clock management. Primarily, the core interfaces between the serialized incoming HT data (in LVDS format) and the standard cHT packets which are exchanged with the logic behind the core. We designed and implemented the custom transport layer logic, the Data Transfer Engine (DTE), to process these packets. The DTE handles: enforcement of protocol-level correctness; piecing together and unpacking HT commands; packing up and sending HT commands; and HT tag management. The DTE also handles all the details of being a coherent node in the system, such as responding to snoop requests. In addition, the FARM platform includes a parameterized set-associative coherent cache. We will provide design and implementation details for the DTE and the cache later in this section. Finally, there is also a small memory mapped register (MMR) file for status checking and other small-scale communication with the processors.

The FARM platform provides three communication interfaces for the hardware being prototyped by the user on the FPGA, or the *user application*: an MMR interface, a stream interface, and a coherent cache interface. A detailed comparison of these interfaces can be found in Section III.

We use dual-clock buffers and (de-)serialization blocks to partition the FPGA into three different clock domains: the HyperTransport links, the cHT core, and the rest of the acceleration logic (everything "above" the cHT core). In our base configuration: the user application and cHT core run at 100 MHz and the HyperTransport links at 200 MHz.

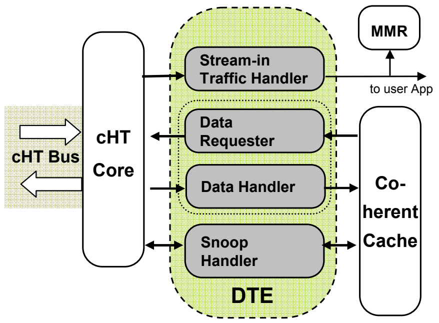


Figure 3. Block diagram of data transfer engine (DTE) components. Arrows represent requests and data buses.

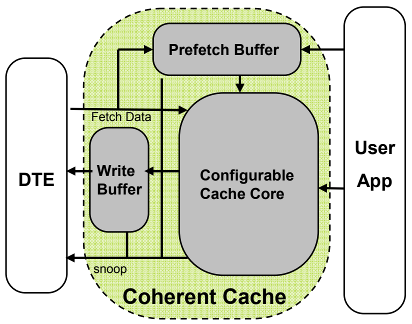


Figure 4. Block diagram of coherent cache components. Arrows represent direction of data flows, rather than that of requests.

B. Module Implementation

The DTE and the cache are two vital units allowing the accelerator to communicate with the processors, process snoops, and store coherent data. In this section, we briefly describe the design and structure of these modules as implemented on our FPGA.

1) Data Transfer Engine: The DTE's primary responsibility is ensuring protocol-level correctness in HyperTransport transactions. Figure 3 shows a block diagram of the components of the DTE. A typical transaction is the following: If the data requester on the FPGA requests data from remote memory (owned by one of the Opteron CPUs), *snoops* and *responses* must be sent among all coherent nodes of the system (assuming no directory) to ensure that any dirty cached data is accounted for. In this example, because the FPGA is the requester, the DTE's data handler is responsible for counting the responses from all caches as well as the data's home memory controller and selecting the correct version of the data. Evictions from the FPGA's cache to remote memory are also fed to the cHT core via the data requester. In addition, snoops incoming to the FPGA are processed by the snoop handler in the DTE. The DTE also handles incoming traffic for stream and MMR interfaces. In doing so, the DTE acts as a pseudo-memory controller for memory requests belonging to the FPGA's memory range. Coherent HyperTransport supports up to 32 simultaneously active transactions by assigning tags to each transaction, so the design must be robust to transaction responses and requests arriving out of order. The DTE handles this by using tag-indexed data structures and tracking tags of incoming and outgoing packets in the data stream interface.

2) Configurable Coherent Cache: Figure 4 shows the block diagram of our coherent cache module. The cache is composed of three major subblocks. The core is where the traditional set-associative memory lookup happens; the write buffer keeps track of evicted cache lines until they are completely written back to memory; and the prefetch buffer is an extended fill buffer to increase data fetch bandwidth. There are three distinct data paths from the cache to the DTE: fetching data, writing data back, and snooping. All data transfers happen at cache line granularity. The user application can request that the cache prefetch a line and read or write to memory using a normal cache interface.

Our normal cache interface supports simple in-order reads and writes at word granularity.[2] This is a valid compromise of design complexity (and power, area, and verification) against application performance since we seldom expect complex out-of-order computation behind our cache. However, the user application can initiate multiple data fetch transfers through the prefetch interface. Unlike the normal interface, the prefetch interface is non-blocking as long as there is an empty slot in the buffer. This design is based on the observation that in many cases the user application can pre-compute a set of addresses to be accessed.

The cache module is responsible for maintaining the coherence of the data it has cached. First, the cache answers incoming snoop requests by searching for the line in all three subblocks simultaneously. Snoop requests have the highest priority since their response time is critical to system-wide cache miss latency. Second, the module must maintain the coherence status of each cached line. For simplicity, our current implementation assumes that cache lines are either modified or invalid; exclusive access is requested for each line brought in to the cache. This simplification is based on the observation that for our current set of target applications, the cache is most often used for producer-consumer style communication where non-exclusive access to the line is not beneficial.

The cache uses physical addresses, not virtual addresses. This saves us from implementing address translation logic, a TLB, and a page-table walker in hardware and from modifying the OS to correctly manage the FPGA's TLB. Instead we rely on the software to use pinned pages provided by our device driver for shared data.

C. FPGA Resource Usage

Table II shows an overview of the resource usage on the FPGA. We made an effort to minimize the usage of FPGA resources by FARM modules in order to maximize free resources for the user application. Note that the cache

[2]In actuality, our cache is not strictly in-order but supports hit-under-miss. That is, the interface stalls at the second miss, not the first.

	FARM modules
4Kbit Block RAMs	144 (24%)
Logic Registers	16K (15%)
LUTs	20K
FPGA Device	Stratix II EPS130
FPGA Speed Grade	-3 (Fastest)

Table II
SUMMARY OF FPGA RESOURCE USAGE.

module has several configuration parameters, including total size and associativity of the cache, size of each cache line, and others. These parameters are configured at synthesis time to meet area, frequency and performance constraints for application. The numbers for *FARM modules* in the table reflect a 4KB, 2-way set associative cache.

III. COMMUNICATION MECHANISMS

FARM supports multiple communication mechanisms tailored for different situations. Applications may use traditional memory-mapped registers (MMRs), a streaming interface for pushing large amounts of data to the FPGA with low overhead, or a coherent cache for communicating with the FPGA as if it were another processor in a shared memory system.

MMRs are traditionally used for infrequent short communication, such as configuration, because of the time required to read and write to them. FARM allows for much faster access to the MMRs because of the FPGA's location as a point-to-point neighbor of the processors. Specifically, we measured the total time to access an MMR on farm to be approximately 672 ns, nearly half the measured 1240 ns to read a register on an ethernet controller directly connected to the south bridge via PCIe x4. This lower latency allows MMRs in FARM to be used for more frequent communication patterns like polling. More detailed measurements show that most of the 672 ns is spent handling the access inside the FPGA, indicating that this latency could be further reduced by upgrading to a faster FPGA.

Currently, FARM's MMRs uses uncached memory, which provides strong consistency guarentees. However, this means that access to multiple MMRs will not overlap and the total access time will grow linearly with the number of registers accesses, just like those to normal PCI registers. With FARM it is just as simple to put the MMRs in the write-combining space, which has weaker consistency guarantees but would allow multiple outstanding accesses (although still disallow caching) and thus provide much faster multi-register access. Section IV uses uncached memory for the MMRs, as the uncached semantics are closer to the expected use of MMRs.

FARM's streaming interface is an efficient way for the CPU to *push* data to the FPGA. To facilitate streaming data, a physical address range marked as write-combining is mapped to the FPGA. Writes to this address range are immediately acknowledged and piped directly to the *user application* module. The internal pipeline passes 64 bits of data and 40 bits of address to the user application per clock.

Service Location of cache miss	FARM	FARM w/o FPGA
Memory	495 ns	189 ns
Other cache (on-chip)	495 ns	145 ns
Other cache (off-chip)	500 ns	195 ns
FPGA cache (1-hop)	491 ns	N/A
FPGA cache (2-hop)	685 ns	N/A

Table III
COMPARISON OF CACHE MISS LATENCY

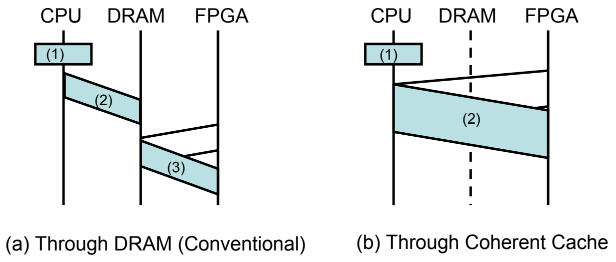


Figure 5. Comparison of DMA schemes.

On the CPU, write requests to the streaming interface are queued in the core's write-combining buffer and execution continues without waiting for the request to be completed. Consecutive accesses to the same cache line are merged in the write-combining buffer, reducing off chip bandwidth overhead. Thus, to avoid losing writes, every streamed write must be to a different, ideally sequential, address. The CPU periodically sends requests from the buffer to the FPGA or an explicit flush can be performed to ensure that all outstanding requests are sent to the FPGA.

Finally, the coherent cache allows for shared memory communication between the CPUs and FPGA. Since the cache on the FPGA is kept coherent, the FPGA can transparently read data either directly from a CPU's cache or from DRAM, and vice versa. The communication latency is simply the off-chip cache miss latency, which is summarized in Table III. In the table, the column labelled *FARM* shows the cache miss latency measured on the current FARM system. Except when the requesting CPU is two hops away from the FPGA, this latency is fairly constant because the FPGA's response to the snoop dominates any other latency. For comparison we also provide measurements using the same system with the FPGA removed. This increase in latency would be intolerable for an end product, but is reasonable for a prototype platform and would be mitigated by using a faster FPGA.

The coherent communication mechanism is especially beneficial when performing a *pull*-type data transfer (i.e. DMA), or when polling for an infrequent event. Figure 5 illustrates two different ways of performing a DMA from the CPU to the FPGA. Figure 5.(a) is the conventional DRAM-based method, where (1) a CPU first creates data in its own cache, (2) the CPU moves the data to DRAM, and (3) the

Interface	Description	Proposed Usage	Aprox. Bandwidth
MMR	CPU writes to FPGA's MMR	Initialization or change of configuration	25 MB/s
MMR	CPU reads from FGPA's MMR	Polling (likely to hit)	25 MB/s
Stream	CPU writes into FPGA's address space	Data push	630 MB/s
Coherent	CPU reads from FPGA's cache	Data pull or Polling (likely to miss)	630 MB/s
Coherent	FPGA reads from CPU's cache (i.e. coherent DMA)	Data pull or Polling (likely to miss)	160 MB/s

Table IV
SUMMARY OF COMMUNICATION MECHANISMS.

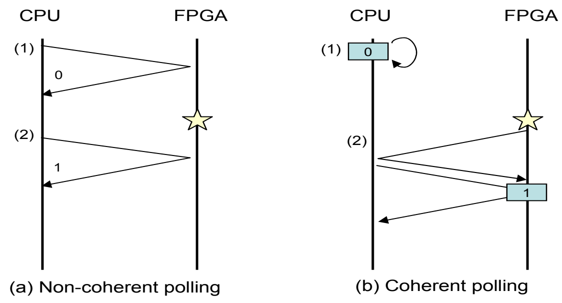


Figure 6. Comparison of non-coherent and coherent polling.

FPGA reads the data from DRAM. Note that during the data preparation steps, (1) and (2), the CPU is kept busy. FARM's coherence allows the method shown in Figure 5.(b), where (1) the CPU leaves the data and proceeds while (2) the FPGA reads the data directly from the CPU's cache.

The coherent interface is also beneficial when polling infrequent events [7]. Figure 6 illustrates this by comparing (a) non-coherent polling through MMR reading and (b) coherent polling through a shared address. In both cases, the event to be polled is represented as a star, and the CPU polls it before and after the event, denoted as (1) and (2) respectively. In Figure 6.(a), (1) and (2) have the same MMR reading latency, while in (b), (1) has the negligible latency of a cache hit and (2) has up to twice the cache miss latency. Thus, when the event is infrequent, the majority of checks performed by the CPU are simply a cache hit and do not stall the CPU at all.

Table IV summarizes communication mechanisms based on FARM's three interfaces and their proposed usages. The MMR bandwidth numbers are for MMRs are in uncached memory. The roundtrip latency to the FPGA is the limiting factor for the MMR bandwidth. The bandwidth of the FPGA reading from the CPU's cache is limited by the bandwidth of the cHT core because the data read pathway has not been optimized. Measurements indicate that optimizing this pathway could bring this number up to at least 320 MB/s.

IV. MICROBENCHMARK ANALYSIS

Designers using FARM systems would benefit from understanding how key application characteristics affect the overhead introduced by the system. For example, it is clear that one would avoid the fully synchronous MMR write for frequent communication with the FPGA. Less obvious, however, is the choice between using streaming versus DMA for moving data to the FPGA. Side effects such as CPU involvement, which would be considerably more for the streaming case, complicate matters further.

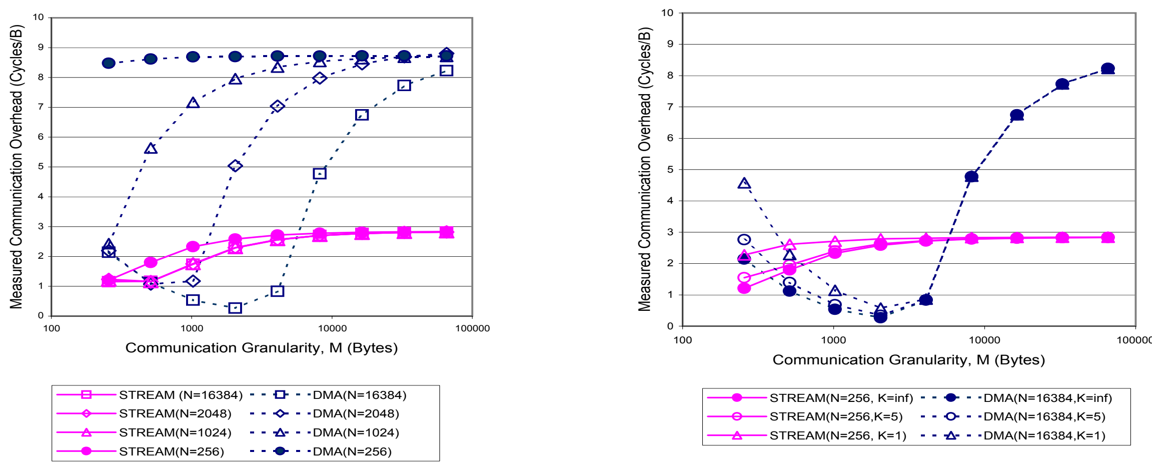
```
main (numIter, commType, N, M, K)
  for i = 1 to numIter
    for j = 1 to K
      InitCommunication(commType, M);
      DoComputation(N);
    Synchronize(commType);

InitCommunication(commType, M)
  switch (commType)
    case MMR:    DoMMRWrite(M);
    case STREAM: DoStreamWrite(M);
    case DMA:    InitiateDMA(M);

DoComputation(N)
  for j = 1 to N
    nop();

Synchronize(commType)
  switch (commType)
    case MMR:
      nop(); // MMR is always synchronous
    case STREAM:
      FlushWriteCombiningBuffer();
    case DMA:
      WaitForDMADone();
```

Figure 7. Microbench for characterizing communcation mechanisms.

To adequately address questions such as these, we constructed a microbenchmark that allows for variation of key parameters affecting communication overhead. Figure 7 displays its pseudocode. Three parameters control the behavior of the communication:

- N controls the frequency of communication. That is, communication happens every N CPU operations.
- M controls the granularity of communication by specifying how much data (in bytes) is transferred per communication.
- K controls the frequency of synchronization. Synchronization occurs after every K sets of communication/computation segments. If K is ∞, we assume synchronization happens only once: at the end of the application.

Figure 8 explores the effects of communication granularity, communication frequency, and synchronization on communication overhead. The vertical axis is communication overhead measured in cycles per byte received by the FPGA (lower is better). We first examine the case of asynchronous communication (i.e. K is ∞) in graph (a).



(a) Effect of communication granularity(M) and frequency(N)

(b) Effect of synchronization frequency (K)

Figure 8. Analysis of communication mechanisms using microbenchmark in Algorithm 7. The detailed meaning of parameter M,N,K can be found there.

For the streaming interface (solid lines), the results for all communication frequencies are asymptotic, with the overhead approaching 2.8 cycles/B for large M. After taking into account the CPU clock frequency (1.8GHz), this value is close to the 630 MB/s bandwidth limit reported in Table IV. As we decrease M, however, we see the overhead decrease and surpass the bandwidth limit. This is because for smaller amounts of data, the overhead can be hidden by the CPU's out-of-order window. Figure 9.(a) provides a visualized explanation of this effect. For frequent communication (N=256), there is not enough computation to hide the communication latency, which explains the increased overhead for this data point compared to the other three.

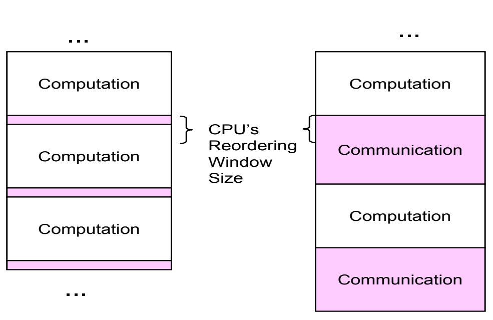
For DMA communication of data from the CPU's cache to the FPGA(dashed lines), we immediately see that the overhead is increased due to the bandwidth explained in Section III. Note, however, that the general behavior of the curves is similar to that of the streaming case. Figure 9.(b) provides further insight into DMA behavior. The figure on the left depicts the case where N=16384 and M=1024. In this scenario, the actual DMA transfer time is fully overlapped with the subsequent computation. When this is the case, the overhead is simply the time taken to setup the DMA. For very small M, the small amount of computation per DMA is not enough to amortize this setup time. As the amount of data per communication goes up, the setup time is amortized and the overhead per byte goes down. If we increase M to the point that data transfer time becomes longer than computation time (seen on the right of Figure 9.(b)), we see a dramatic increase in the overhead. As in the streaming case, the overhead converges to the bandwidth of the DMA transfer (See Table IV).

Figure 8.(b) explores the effect of synchronization frequency. Smaller K means more frequent synchronization. We take two data points from graph (a) for both streaming (N=256) and DMA (N=16384), and we vary K. For the streaming interface, synchronization means flushing of the write-combining buffer. For coherent DMA, synchronization requires waiting (busy wait) until all queued DMA operations have finished. For very large communication granularity (M), the overhead is bounded by the bandwidth in both cases and synchronization does not matter. For smaller M, however, both communication methods exhibit an increase in overhead. For the streaming interface, flushing the write buffer cripples the CPU's out-of-order latency-hiding effect, hence the increased overhead for K=1. For DMA, synchronization adds the fixed overhead of setting up the DMA.

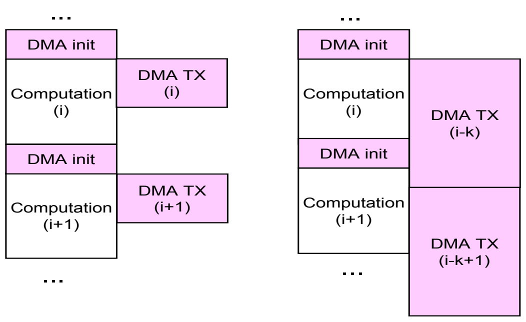
V. Potential Applications

In this section we briefly explore what other applications could be efficiently prototyped with FARM. We have already used FARM to prototype a software transactional memory accelerator that requires the use of FARM's low-latency fine-grained communication to track accesses to shared data. We therefore assert that FARM can also be used for similar applications that require fine-grained communication, such as hardware-assisted data race detection [8] or runtime profiling [9].

While FARM does not have the freedom to change the processor, it will still be possible to design intelligent peripheral devices. Coherent access to the CPU's cache can simplify the design of previous intelligent I/O devices such as Underwood et.al.'s accelerator for MPI queue processing [10] (see also Mulkherjee et.al.'s work [7]). FARM could also be used to prototype intelligent memory systems for performance [11] or for security [12]. Such a system would extend the memory controller described in Section II-B1



(a) Stream interface's case



(b) DMA interface's case

Figure 9. Visualized explanation of graph 8. For stream interface's case, when granularity(M) is large the communication overhead is solely determined by the bandwidth limit, while CPU's instruction reordering can hide it for small M. Similar explanation applies to DMA's case, where communication overhead can be completely hidden depending on the choice of M and N.

with the required intelligence using additional information available through the coherent interface.

In addition, FARM can help prototype advanced coherent protocols. For example, one could prototype directory structures such as Acacio et.al.'s [13], or snoop filtering techniques like Moshovos' RegionScount [14]. Note that such extensions of the underlying broadcasting coherence protocol (cHT) have been proposed in the original design [15] but actual implementations have been rare.

Finally, FARM also benefits the traditional computation-oriented FGPA applications like FFT computation or matrix multiplication, either by reduced communication latency or convenient communications interfaces.

VI. Conclusion

In this paper we presented FARM, a hardware prototyping system based on an FPGA coherently connected to multiple processors. In addition, we described and characterized several different mechanisms for communicating with the FPGA in FARM. FARM provides tools that enable researchers to prototype a broad range of interesting applications that would otherwise be expensive and difficult to implement in hardware.

VII. Acknowledgements

This work was supported by the Stanford Pervasive Parallelism Lab, Dept. of the Army AHPCRC W911NF-07-2-0027-1, National Science Foundation CNS0720905, DOE/Sandia 942017, and DARPA/Sun 630003198.

We also thank A&D Technology, Inc. for donating the Procyon systems to us and for their excellent support.

References

[1] L. Barroso, S. Iman, and J. Jeong, "RPM: A rapid prototyping engine for multiprocessor systems," *IEEE Computer*, 1995.

[2] J. Wawrzynek *et al.*, "Ramp: Research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, 2007.

[3] C. C. Corp. Instruction set innovations for convey's hc-1 computer. [Online]. Available: http://www.conveycomputers.com/Resources/Convey.Hot%20Chips.Brewer.pdf

[4] I. AMD. Maintaining cache coherency with amd opteron processors using fpga's. [Online]. Available: http://ra.ziti.uni-heidelberg.de/coeht/pages/events/20090211/parag_beeraka.pdf

[5] U. of Heidelberg (Germany). UoH cHT-Core (coherent HT Cave Core). [Online]. Available: http://www.hypertransport.org/default.cfm?page=ProductsViewProduct&ProductID=84

[6] A & D Technology, Inc. Procyon, the ultra-high-performance simulation and control platform. [Online]. Available: http://www.aanddtech.com/products/realtime_measurement_simulation_control/procyon/

[7] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood, "Coherent network interfaces for fine-grain communication," in *ISCA '96: Proceedings of the 23rd Annual Int'l Symp. on Computer Architecture*, 1996, pp. 247–258.

[8] P. Zhou, R. Teodorescu, and Y. Zhou, "Hard: Hardware-assisted lockset-based race detection," in *HPCA '07: Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[9] C. B. Zilles and G. S. Sohi, "A programmable co-processor for profiling," in *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001, p. 241.

[10] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell, "A hardware acceleration unit for mpi queue processing," in *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005, p. 96.2.

[11] C. J. Hughes and S. V. Adve, "Memory-side prefetching for linked data structures for processor-in-memory systems," *J. Parallel Distrib. Comput.*, vol. 65, no. 4, pp. 448–463, 2005.

[12] P. Kocher, R. Lee, G. McGraw, A. Raghunathan, and S. Ravi, "Security as a new dimension in embedded system design," in *Design Automation Conference, 2004. Proceedings. 41st*, 2004, pp. 753–760.

[13] M. E. Acacio, J. González, J. M. García, and J. Duato, "A new scalable directory architecture for large-scale multiprocessors," in *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001, p. 97.

[14] A. Moshovos, "Regionscout: Exploiting coarse grain sharing in snoop-based coherence," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005.

[15] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, no. 2, pp. 66–76, 2003.