2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines

LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories

Hsin-jung Yang[¶] Kermin Fleming[†] Michael Adler[†] Joel Emer[†¶]

[¶]Massachusetts Institute of Technology, CSAIL
{hjyang, emer}@csail.mit.edu

[†]Intel Corporation, VSSAD Group
{kermin.fleming, michael.adler, joel.emer}@intel.com

***Abstract*—Parallel programming has been widely used in many scientific and technical areas to solve large problems. While general-purpose processors have rich infrastructure to support parallel programming on shared memory, such as coherent caches and synchronization libraries, parallel programming infrastructure for FPGAs is limited. Thus, development of FPGA-based parallel algorithms remains difficult. In this work, we seek to simplify parallel programming on FPGAs. We provide a set of easy-to-use declarative primitives to maintain coherency and consistency of accesses to shared memory resources. We propose a shared-memory service that automatically manages coherent caches on multiple FPGAs. Experimental results of a 2-dimensional heat transfer equation show that the shared memory service with our distributed coherent caches outperforms a centralized cache by 2.6x. To handle synchronization, we provide new lock and barrier primitives that leverage native FPGA communication capabilities and outperform traditional through-memory primitives by 1.8x.**

***Keywords*—FPGA shared memory; coherency; synchronization**

I. Introduction

Parallel programming is enormously popular in many fields of science and engineering, including fluid dynamics, geometric modeling, and image processing. This popularity is due, in part, to the strong system-level support and powerful abstractions for parallel programming provided by general-purpose architectures. Threading libraries enable the light-weight partitioning of programs, and the shared-memory programming model provides these threads a single view of memory. In addition, synchronization libraries help software programmers coordinate computation among the threads.

Abundant parallelism and fast communication make FPGAs attractive for accelerating algorithms that traditionally run on general-purpose hardware, including parallel programs. Although FPGAs have great potential, they have traditionally been viewed as difficult to program. Recently, researchers have been working on providing new programming primitives, such as communication [1] [2] and memory abstractions [3], to shorten the development time of FPGA programs and make FPGAs easier to use. Recent advances in FPGA compilation techniques allow FPGA programmers to easily partition designs across multiple FPGAs [2]. Together these advances have enabled the construction of large systems of FPGAs. Such systems have traditionally been ideal targets for parallel programming.

Despite the potential of FPGAs, mapping parallel algorithms to FPGAs remains difficult. Although FPGAs intrinsically support parallel program descriptions, good libraries for shared memory and synchronization are not available, especially in the context of logic programming. As a result, FPGA programmers usually adopt a distributed (non-shared) memory model and explicitly handle all data sharing between processing engines [4] [5]. This approach prolongs development time because programmers are fully exposed to the complexity of distributed coordination.

In this work, we seek to simplify FPGA-based parallel programming by providing a set of easy-to-use declarative primitives to maintain coherency and consistency of accesses to shared memory resources. We propose the coherent scratchpad, which manages memory coherency while presenting users with a simple interface similar to FPGA on-die SRAM blocks. Coherent scratchpads provide the illusion of unlimited virtual storage, while automatically managing multiple coherent caches and multiple coherence domains across multiple FPGAs. Since coherency comes with a need for consistency, the coherent scratchpad interface also provides non-blocking memory fences that enable the support of various consistency models in the FPGA. Our coherence scheme automatically scales across FPGAs because it makes use of high-level communications primitives for which sophisticated compilers are available.

Distributed coordination is fundamental in shared-memory programs. In general-purpose processors, locks and barriers are often handled through memory, since memory is usually the only mechanism available for communication. In FPGAs, other direct communication mechanisms are available. Thus, it is more efficient on FPGAs to provide our lock and barrier services outside of shared memory.

We evaluate the performance of our coherency and synchronization primitives using multiple benchmarks which we map to both single and multiple FPGAs. Our proposed primitives enable the concise description of parallel algorithms: these benchmarks require around 350 lines of codes and were written in only a few hours, which we argue is a substantial reduction in code complexity and development time. For our 2-dimensional heat transfer benchmark, coherent scratchpads with coherent caches provide up to 3.8x speed-up over the single-engine baseline and run 2.6x faster than a centralized-cache implementation. The proposed synchronization service outperforms the traditional through-memory primitives by 1.8x in the shared-queue benchmark, and our barrier primitive also achieves 340x higher throughput than that of an existing FPGA barrier primitive.

II. Background

Our shared-memory programming primitives build upon two prior declarative primitives for memory and communication within the FPGA: LEAP Scratchpads [3] and latency-insensitive channels [2]. We leverage the primitives and

978-1-4799-5111-6/14 $31.00 © 2014 IEEE
DOI 10.1109/FCCM.2014.43

```
interface MEM_IFC#(type t_ADDR, type t_DATA);
    method void readRequest(t_ADDR addr);
    method t_DATA readResponse();
    method void write(t_ADDR addr, t_DATA data);
    // t_REQ r := {READ, WRITE, FULL}
    method void fence(t_REQ r);
    method Bool requestPending(t_REQ r);
endinterface
```

Fig. 1: A general memory interface for hardware designs. Our extensions to this interface are highlighted.

compilation support provided in these works to simplify the design of our new primitives.

LEAP Scratchpads provide a general, in-fabric memory abstraction for FPGA programs. Programmers instantiate memories with the simple read-request, read-response, write interface, shown in Figure 1. Each instantiated memory represents a logically private address space, and a program may instantiate as many memories as needed. Memories may have arbitrary size, even if the target FPGA does not have sufficient physical memory to cover the entire requested memory space.

At compile time, the compiler gathers the scratchpads in the user program and instantiates a complex memory hierarchy [3] with multiple levels of cache, as in Figure 2. Scratchpad memories instantiated in the user program optionally receive a local L1 cache. The board-level memory, typically an off-chip SRAM or DRAM, is used as a shared L2 cache. The L1 caches are connected to the L2 by way of a compiler-synthesized interconnect network. The main memory of an attached host processor backs the synthesized cache hierarchy. Like memory hierarchies in general-purpose computers, scratchpads provide the appearance of fast memory to programs with good locality, while maintaining the abstraction of a large address space through the virtual memory mechanisms of the host.

Our coherent distributed memory primitives export the same memory network interface as the original scratchpads. Thus, we can integrate our new coherent caches directly into the scratchpad memory hierarchy.

Latency-insensitive channels are a recently proposed communications primitive for RTLs [2]. Latency-insensitive channels have operational behavior similar to FIFOs, but may have dynamically variable latency and buffering, allowing the programmer to directly describe points in a program where the compiler *may* choose to alter the timing behavior of the system. For example, a compiler may automatically instantiate a complex inter-chip network if the endpoints of a latency-insensitive channel are on two different FPGA chips. Programs framed in terms of latency-insensitive channels may be targeted to any configuration of FPGAs, including a single FPGA, without modifying the user source code.

In this work, we describe the implementation of all of our shared memory primitives in terms of latency-insensitive channels. We then leverage the compiler described in [2] to partition designs using our primitives across different configurations of FPGAs.

III. Memory Coherency and Consistency

LEAP Scratchpads are a powerful abstraction for describing private, independent memory spaces. However, scratchpads are insufficient as a shared memory programming substrate because programmers are responsible for managing all sharing. For example, a user may instantiate a single large programmer-multiplexed scratchpad that connects to multiple processing engines. This kind of implementation is undesirable because it scales poorly: a single memory forces the serialization of requests. Moreover, if a processing engine is on a remote FPGA, the inter-FPGA latency, which is at least an order of magnitude longer than the intra-FPGA latency, increases the memory access latency and may have great impact on performance. In this work, we solve these performance issues by introducing a distributed, coherent cache local to each accessor. We extend the baseline scratchpad to manage a coherent cache and refer it as a *coherent scratchpad* (CS). Henceforth, we refer to the scratchpad design that provides only private memory as a *private scratchpad* (PS).

Coherent scratchpads retain all the properties of private scratchpads: arbitrary data size, virtualization, and a simple user interface. Like the shared-memory abstraction in general-purpose machines, the coherent scratchpads are largely transparent to programmers, enabling programmers to focus on designing parallel algorithms.

A. Coherent Scratchpad Interface

Our coherent scratchpads support the private scratchpad interface: `readRequest`, `readResponse`, and `write`. In addition, we provide a new consistency interface. As in processor shared memory, fences are necessary because the coherent scratchpad implementation serves memory requests out-of-order to improve performance. We provide three kinds of fences: write, read, and full fences to support various consistency models, allowing programmers to manage program ordering. A full fence ensures that all read and write requests prior to the fence will be processed before any request that comes after the fence is processed. Similarly, a write fence enforces the ordering of write requests issued before and after the fence request, and a read fence enforces the ordering of read requests. All memory requests, including reads, writes, and fences, are pipelined, and users are allowed to issue read or write requests after fence requests to maintain pipeline parallelism. Since supporting fences adds additional logic and latency to the scratchpad pipeline, we also provide a second, coarser consistency mechanism: a `requestPending` signal indicating whether there is an incomplete request.

While general-purpose processors usually assume a single coherence domain in which all threads share a single global memory, we take advantage of FPGA flexibility and allow FPGA users to specify multiple coherence domains with independent memory address spaces. Disjoint coherence domains do not interact. The following example shows how to specify a coherence domain and instantiate associated coherent scratchpads:

```
// Instantiate a coherent scratchpad controller
mkCohScratchController(domainID, addrSize, dataSize);
// Instantiate coherent scratchpad clients
let client1 <- mkCohScratchClient(domainID);
let client2 <- mkCohScratchClient(domainID);
let client3 <- mkCohScratchClient(domainID);
```

B. Coherence Protocol

LEAP Coherent Scratchpads specify an abstract shared memory interface which can be backed by any coherence protocol. Coherent scratchpads currently implement a snoopy protocol. Unlike general-purpose processors that typically

support only one coherence domain across all cores, coherent scratchpads support multiple coherence domains, separating FPGA programs that touch different address spaces. In addition, programs that do not require coherence can simply use private scratchpads. This limits the number of clients of a memory resource to the exact number of those who actually need to share data and consequently limits the performance impact of the snoopy protocol.

Coherent scratchpads in the same coherence domain are connected via ring networks and use a global ordering point to ensure that all coherent scratchpads see the coherence requests in the same order [6]–[8]. Coherence requests are sent to the global ordering point where they are activated. The ordering point then broadcasts these activated requests on the ring. Each coherent scratchpad and the backing, next-level memory snoop the activated requests, taking actions depending on the local coherence state of the target address.

We implement a MOSI protocol which is based on the protocol specified in gem5 [9]. A cache line is in the M (modified) state if the cache block is dirty and no other caches have a valid data copy. A cache line is in the O (owned) state if the cache is one of the several with a valid dirty copy of the data block and it is the only one that owns the data. If the cache line is in the M or O state, then it is the owner of the data block and is responsible for writing back dirty data to memory if the line gets evicted. In addition, the protocol supports data forwarding, which means the block owner is responsible for responding to other caches' snoop requests. A cache line is in the S (shared) state if it is one of the several containing a valid copy of the data block, which may be either dirty or clean. A cache line is in the I (invalid) state if the cache does not have a valid copy of the line. To perform a write operation, the cache line must be in the M state, while read can be performed if the cache line is in the M, O, or S state. The next-level memory keeps a directory that stores an owner-bit for each memory address, indicating whether the data is owned by one of the coherent caches or not. The directory, combined with the cache ownership information, ensures that only the actual owner of the data responds to a snoop request, without requiring fine-grained coordination among the controller and caches.

To optimize read-modify-write and to simplify the controller, I state is automatically upgraded to M state on a first read to the next-level memory. In this way, the subsequent write can be performed directly without notifying other caches. However, this means the new M state (and the O state that is transited from the new M state) may contain clean data. To eliminate unnecessary data write-backs, each cache line has an additional dirty bit to support clean write-back, in which only the ownership information is written back to the controller. Our M state can be seen as a combination of the M state and the E (exclusive) state in the standard MOESI protocol.

C. Coherent Scratchpad Architecture

Figure 2 shows the coherent scratchpad architecture, which is integrated into the existing private scratchpad memory hierarchy. In this example, there is one coherence domain that contains three coherent scratchpad clients and one coherent scratchpad controller. A private scratchpad client is also included in the figure for comparison. The coherent scratchpad controller serves as a global ordering point as well as the interface to the next-level memory. To prevent deadlocks,

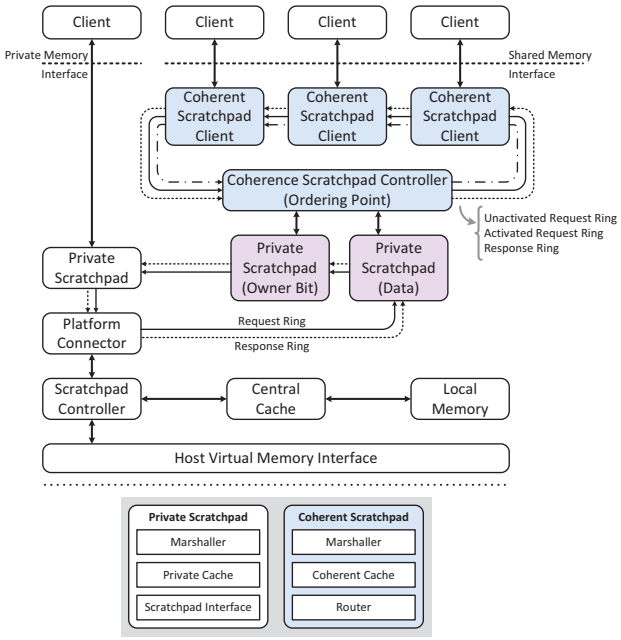


Fig. 2: Coherent scratchpad architecture

coherent scratchpad clients and the controller within the same coherence domain are connected via three rings: the unactivated request ring, the activated request ring, and the response ring. As mentioned in Section III-B, the next-level memory must store both data and the owner-bit for each memory address. To simplify the design, these stores are themselves implemented as private scratchpads. This gives coherent scratchpads high scalability in terms of address space size, since we can leverage the existing memory hierarchy and its virtual memory support to store coherence ownership. Since owner-bit access usually has high temporal locality, utilizing the private scratchpad cache hierarchy makes the coherent memory system more efficient.

D. Client Microarchitecture

As shown in Figure 2, a coherent scratchpad client contains a marshaller, a coherent cache, and a router. The line size of the coherent cache is parameterized, and the client request size is constrained to be smaller than that of a cache line. If the request size is smaller than the line size, a marshaller is instantiated to handle partial reads and writes using masks.

Figure 3 shows the microarchitecture of the direct-mapped coherent cache and router. The cache is designed to serve multiple local requests from the client and network requests from remote caches. A completion table is added in the router to store the metadata of network requests that need to be snooped. The size of the completion table controls the number of snoop requests allowed to enter the cache pipeline. Each component in the cache is pipelined to achieve high throughput, and we allow requests to different memory addresses to be served out-of-order.

To improve throughput, coherence network transactions are split-phase. Decoupling requests and responses increases the complexity of handling cache misses, since there might be multiple outstanding requests targeting the same address. Transient states must be handled during the transition from one

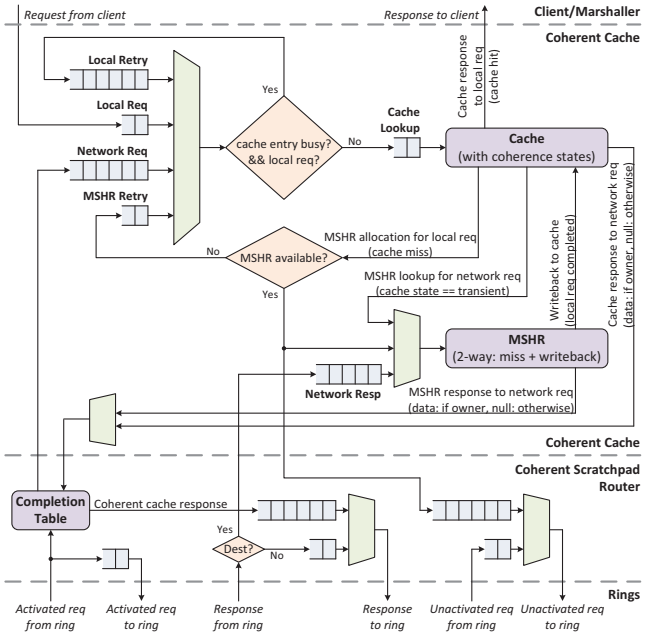


Fig. 3: Coherent cache microarchitecture

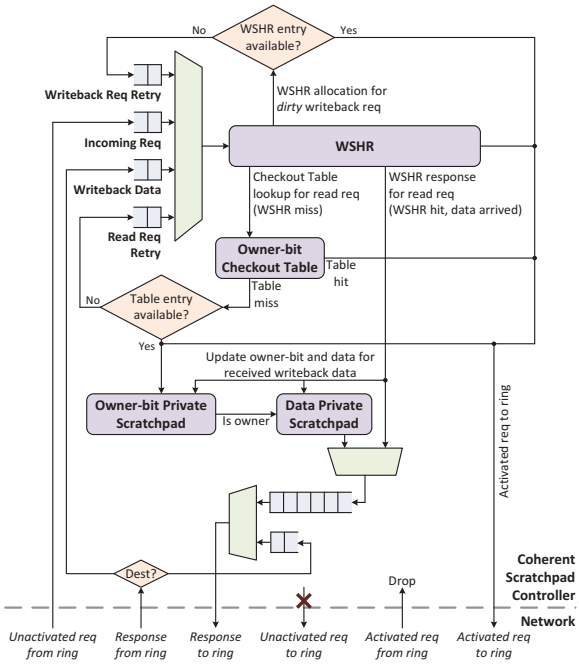


Fig. 4: Coherence controller microarchitecture

steady state to another. We let the cache store the coherence steady states and allocate a 32-entry 2-way miss status handling register (MSHR) table to handle the transient states for cache misses and write-backs. The MSHR maintains a forwarding list to optimize the case where the cache receives a snoop request with the same address as a pending local write request that is still waiting for the data response. The MSHR records the snoop request in the forwarding list, allowing subsequent requests to be served. The recorded requests can be served as soon as the local write is completed.

E. Controller Microarchitecture

The coherence controller shown in Figure 4 serves as the ordering point and the interface to the next-level memory. The controller snoops every coherent request and responds to the request if the line is not owned by one of the cache clients. The controller uses two private scratchpads: one is the data scratchpad, which serves as the interface to the next-level memory hierarchy; the other is the owner-bit scratchpad, which stores the ownership information per memory address.

To reduce the bit-width of the request channel, we separate data and ownership messages for write-backs. The controller contains a write-back status handling register (WSHR) table to track incomplete dirty write-back requests, allowing non-atomic dirty write-backs. As with the MSHRs at the client, the WSHR maintains forwarding lists. If there is a read miss request from one of the caches requesting the same address as an incomplete write-back, the WSHR records the read request and responds to the requester after the write-back data arrives.

F. Deadlock Freedom

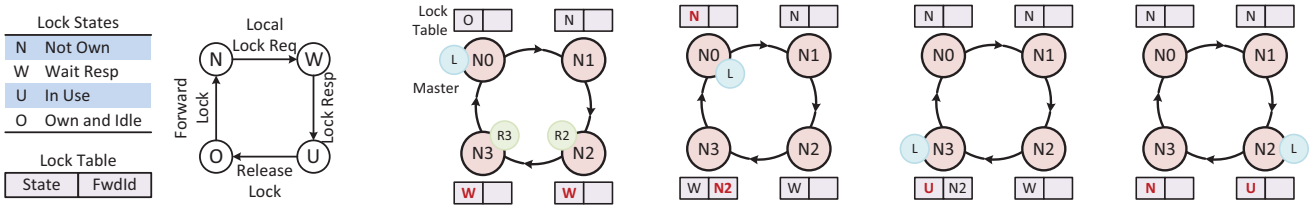
Deadlocks are a classic problem of coherence protocols. We assume that gem5's MOSI protocol is deadlock free. To ensure deadlock-freedom in our implementation, we need to first guarantee that coherent caches do not stall due to insufficient buffer space in the network. To achieve this, our implementation requires that the cache reserve request buffer slots inside its local router before it begins processing a local request. Similarly, an entry is reserved in the router's completion table for each activated request from the network before the request is processed. These reservation mechanisms guarantee that coherent caches can always consume incoming packets from the network. Since the coherent cache is never stalled, a response received from the network can always be consumed and hence does not block the messages from the response channel. At the network level, each message class is transmitted on an independent ring. No messages traverse the same channel twice, avoiding deadlocks in the logical network.

Coherent scratchpads' logical network is implemented with latency-insensitive channels. We rely on a compiler to implement a physical network for these channels, especially for the channels that cross FPGA boundaries [2]. The network implementation produced by the compiler allocates virtual channels in a manner that gurantees deadlock freedom.

IV. Synchronization

In addition to memory coherency and consistency management, parallel programming in shared memory systems also requires synchronization. Locks and barriers are common synchronization primitives. Locks limit access to shared resources, ensuring that no two accessors can enter a critical section at the same time. Only the accessor that obtains a lock has access to the associated shared resource. Barriers pause accessors until all accessors arrive at the barrier.

In general-purpose processors, inter-processor communication typically occurs through memory, and synchronization primitives are usually implemented using atomic memory operations. However, handling synchronization primitives through shared



(a) Lock state (b) Time = T0 (c) Time = T1 (d) Time = T2 (e) Time = T3

Fig. 5: A lock service walkthrough example. At time T0, the master node (N0) owns the lock, and both nodes N2 and N3 issue a get lock request to the network. At time T1, N0 receives request issued by N3 and sends out the lock as response, while N3 records N2 in its lock table. At time T2, N3 receives the lock and starts the critical section. At time T3, N3 finishes its task, releases the lock, and sends the lock to N2.

memory introduces coherence traffic, which is inherently expensive. Since FPGAs offer many communication mechanisms, we have more choices for the implementation of synchronization primitives. LEAP implements lock and barrier services outside of the coherent scratchpad framework. For hardware-based applications, having separated lock and barrier services reduces design area and improves performance.

A. Lock Service

Our lock service supports multiple lock groups that are instantiated using namespaces, similar to the definition of coherence domains. Locks may be shared among any number of clients and multiple locks may be defined within a lock group. Although lock nodes have equal priority, we opt for a master-slave architecture to simplify the initialization of lock state at reset. Each lock group requires a master node. In the following example, four lock nodes that share a single lock are instantiated:

```
// Instantiate four lock nodes (LOCK_NUM == 1)
LOCK_IFC#(LOCK_NUM) n0 <- mkLockNode(groupID, MASTER);
LOCK_IFC#(LOCK_NUM) n1 <- mkLockNode(groupID, SLAVE);
LOCK_IFC#(LOCK_NUM) n2 <- mkLockNode(groupID, SLAVE);
LOCK_IFC#(LOCK_NUM) n3 <- mkLockNode(groupID, SLAVE);
```

The lock service provides an interface as follows:

```
interface LOCK_IFC#(type t_LOCK_ID);
    method void acquireLockReq(t_LOCK_ID id);
    method t_LOCK_ID lockResp();
    method void releaseLock(t_LOCK_ID id);
endinterface
```

A client calls `acquireLockReq` to attempt acquiring a lock. `lockResp` grants the client control of the lock. The client calls `releaseLock` when finishing the access.

Inside each lock group, lock nodes are connected via rings, and the master node of the group initially owns all the locks. Each lock node has a lock table that records a 2-bit lock state per lock. To improve performance, lock nodes also record lock forwarding information. Figure 5 walks through an example to demonstrate how the distributed lock service works.

B. Barrier Service

Programmers use barriers to synchronize multiple concurrent tasks. Similar to lock groups, multiple barrier groups can be instantiated using namespaces. Our barrier service is centralized: each group of barrier nodes requires a master node. The master node is responsible for collecting barrier status from the slave nodes and broadcasting a completion signal. The following example shows how to instantiate two barrier nodes sharing one barrier:

```
// Instantiate two barrier nodes
BARRIER_IFC n0 <- mkBarrierNode(groupID, MASTER);
BARRIER_IFC n1 <- mkBarrierNode(groupID, SLAVE);
```

The barrier service provides the following interface:

```
interface BARRIER_IFC;
    method Bool initialized();
    method void setBarrier(t_BARRIER barrier);
    method void barrierReached();
    method void waitForSync();
endinterface
```

During initialization, the master node sets the synchronization condition (`setBarrier`), specifying which nodes must reach the barrier point before the barrier is completed. Then, the master broadcasts an initialization signal to inform all slave nodes that they can start performing their tasks. When a slave node finishes (`barrierReached`), it sends a message to the master and waits for the barrier completion signal (`waitForSync`). The master node updates its barrier state upon receiving messages from slave nodes. When the barrier is completed, the master node broadcasts a completion signal, notifying the slave nodes that they can start their next step.

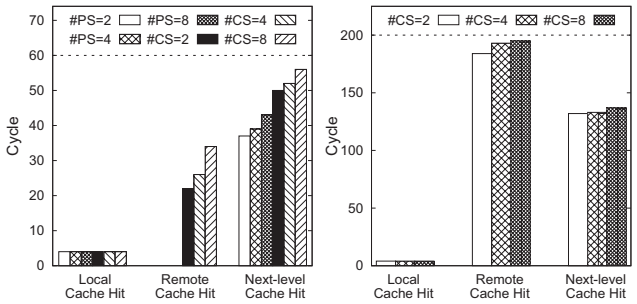
Our barrier implementation resembles the lock service implementation. The nodes that need to be synchronized are connected via rings to form a barrier group. The master node maintains a table tracking barrier states.

V. Evaluation

To evaluate the performance of our coherent scratchpads and synchronization primitives, we target a set of benchmarks to both single and dual FPGA configurations. We use Xilinx VC707 FPGA boards as our evaluation platform. For dual FPGA configurations, we network the two boards using the high-speed SERDES provided by the FPGA fabric. If not specified, we use the following configuration to run the experiments: (1) each private scratchpad (PS) has a 1024-entry 64-bit direct-mapped cache; (2) each coherent scratchpad (CS) has a 1024-entry 64-bit direct-mapped cache and a 32-entry 2-way MSHR; (3) only one next-level cache is used in the dual FPGA tests.

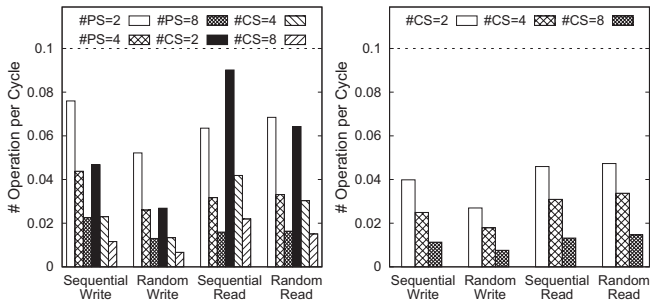
A. Coherent Memory Service

1) Synthetic Benchmarks: To understand the overhead introduced by coherency management, we first measure the latency



(a) Single FPGA (b) Dual FPGA

Fig. 6: Scratchpad hit latency comparison.



(a) Single FPGA (b) Dual FPGA

Fig. 7: Scratchpad throughput comparison.

(Figure 6) and throughput (Figure 7) of systems with various number of coherent scratchpads using synthetic benchmarks. Coherent scratchpads have the same local cache hit latency as private scratchpads, but their hit latency to the next-level cache is longer. This overhead results from the controller checking the ownership information of the requested data block and then forwarding the request to the private scratchpad ring (see Figure 2). In the dual FPGA configuration (Figure 6b), the measured remote cache latency is higher than the next-level cache latency, because in this case the coherence protocol requires more traversals of the slow inter-FPGA link when data is cached remotely.

Figure 7 shows the throughput of various configurations of coherent caches. In the throughput tests, coherent scratchpads access different regions in the shared address space, eliminating coherency traffic. The coherent scratchpad's write throughput is much lower than the private scratchpad's write throughput because coherent caches must first obtain data ownership before writes can be processed, which requires an extra transaction. The read throughput is similar to that of private scratchpads, because the overhead of snooping is small.

2) Heat Transfer Equation: We evaluate the performance of coherent scratchpads using a 2-dimensional heat transfer equation, which is a simple example of stencil computation. In stencil computations, each grid point is repeatedly updated with a function of its neighboring points in both time and space. Stencil programs have regular spatial locality and abundant parallelism; therefore, they are often used as benchmarks to evaluate on-chip parallelism and the performance of memory subsystems. Furthermore, stencil applications are trivial to parallelize if shared memory abstractions are available.

Figure 8 shows how the single-engine baseline performs the serialized 2-dimensional heat transfer computation. Since the pixel values at time `t` only depend on the values at time `t-1`, we allocate two frame buffers in memory: one to hold the current timestep and another to hold the previous timestep. To take advantage of locality and make the program cache-efficient, we interleave the two frame buffers and store them in row-major order. In addition, we decouple memory reads from pixel computation and memory updates, exploiting pipeline parallelism to overlap the memory access latency.

To parallelize the heat transfer computation, the frame is divided into blocks. Each block is assigned to a processing engine, and the grid points at the borders of blocks are shared

```
for t = 0 to T-1
  for y = 1 to N
    for x = 1 to M
      U[t+1,x,y] = C0·U[t,x,y]+Cx·(U[t,x-1,y]+U[t,
                   x+1,y])+Cy·(U[t,x,y-1]+U[t,x,y+1])
    end
  end
end
```

Fig. 8: A simple nested loop that performs the 2-dimensional heat transfer from time 0 to time T on the MxN grid. C_0, C_x, C_y are constants related to thermal diffusivity.

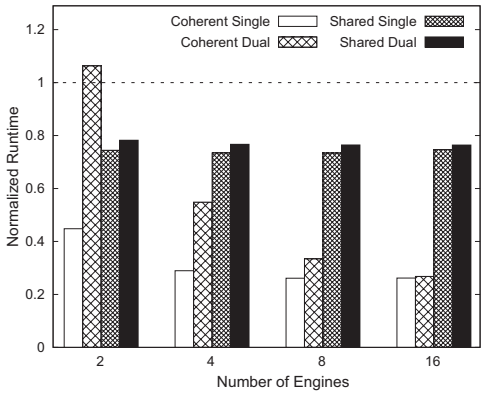


Fig. 9: Heat transfer results. The execution time is normalized to the single-engine implementation using a private scratchpad.

between engines. Each engine includes a coherent scratchpad memory interface to the frame buffers. During an iteration, each engine sweeps through the assigned block, computes, and updates new values until it reaches the barrier point. Then it waits for the barrier signal, which indicates that all engines have completed the current time step, before it starts the iteration of the next time step.

To illustrate the value of coherent caching, we implement a second, uncached configuration where the coherent caches are replaced with an implementation that forwards clients' requests to a centralized, shared scratchpad. We measure the execution time of running the heat transfer equation on a 512-by-512 grid of points for 128 time steps using coherent scratchpads and the shared scratchpad on both single and dual Xilinx VC707 boards. In these experiments, each cache line can fit up to 8 grid points.

TABLE I: SHARED QUEUE TEST RUNTIME RESULT

Primitive	Runtime (Cycles)
Global Arbiter	62043
Hardware Lock	74719
Through-Memory Software Lock	135867

TABLE II: THROUGHPUT COMPARISON OF BARRIER SERVICES

System	Barriers per Second
Our Barrier Service	7352076
Barrier via Coherent Scratchpad	85088
Spin-Lock Mutex-Enabled Cache [10]	21510

To make a fair comparison, we scale the cache size of the shared scratchpad as the number of engines increases. In this way, the total cache size of the coherent scratchpad configuration is the same as the shared scratchpad's cache size. Figure 9 shows the runtime comparison. The execution time of each test is normalized to the execution time of the single-engine baseline implemented with a single private scratchpad. Coherent caches are successful in exploiting temporal and spatial locality, providing up to 3.8x speed-up when there are more processing engines, while the shared scratchpad configuration only gives a 1.4x speed-up. The execution time of coherent scratchpads on dual FPGAs is slower than coherent scratchpads on a single FPGA, because inter-FPGA latency causes a higher miss penalty. In particular, the dual FPGA configuration is much slower when there are only two processing engines, because two engines do not generate enough pipelined traffic to cover the inter-FPGA latency.

B. Synchronization Service

To evaluate the efficiency of our synchronization services, we compare the performance of a shared queue test that uses our lock and barrier primitives with two alternate implementations. One configuration implements locks and barriers using accesses to a set of coherent scratchpads. To support through-memory software locks and barriers we extend the interface in Figure 1 to include an atomic `testAndSet` operation. The other configuration uses a wire-based global arbiter to coordinate exclusive accesses and synchronization. Because this implementation does not communicate through latency-insensitive channels, it is limited to a single FPGA.

We test our three implementations using a simple shared queue benchmark. In this test, a single queue is shared among several producers and consumers, which contend for the ownership of the tail and head of the queue. Upon obtaining the producer/consumer lock, the producer/consumer performs one operation and releases the lock. Table I shows the runtime of a 64-entry shared queue with two producers and two consumers. Each producer has 1024 items to insert. Compared to the wire-based implementation, our lock and barrier services introduce 20% overhead, while the implementation with through-memory software locks and barriers is much slower due to extra coherency transactions.

In addition to the shared queue test benchmark, we also compare the throughput of our barrier service with a through-memory barrier implementation and an existing FPGA-based barrier system, which is implemented using mutexes in coherent soft processors [10]. The three barrier systems are evaluated with 1000 iterations on 8 threads. The through-memory barrier implementation runs at 110 MHz clock frequency, and the other two run at 125 MHz. Table II shows the barrier throughput comparison. The throughput of our hardware barrier primitive is 86x higher than that of the through-memory software barrier and 340x higher than that of the existing work. This demonstrates that it is much more efficient to build barriers outside of shared memory on FPGA. In addition, the through-memory barrier system using coherent scratchpads achieves 4x throughput advantage compared to the existing system. The throughput difference may come from the overhead of running instructions on soft processors and the performance differences between the two coherent cache systems.

C. Implementation Area

Table IIIa lists the maximum place-and-route frequencies and implementation areas of our shared-memory primitives. The synchronization primitives require minimal area to implement. The coherent scratchpad client uses approximately three times the slice LUTs of the baseline private scratchpad. This size increase is not a surprise: coherence protocols are complicated. In addition to the extra slice resources, the coherent scratchpad client uses more BRAMs as compared to the baseline client. The extra BRAM usage comes from the need to store coherence status bits in addition to the baseline cache metadata.

Our coherent scratchpad clients as well as controllers are highly parametric. Table IIIb explores the effect of cache parameters on physical implementation. Since most of the miss handling logic is constant regardless of cache size, increasing the cache size has very little impact on the area usage of the cache. Interestingly, BRAM usage does not scale linearly with cache size. This is because increasing the cache size decreases the cache metadata size such that entire cache lines can fit in a single BRAM word.

VI. RELATED WORK

Distributed memory management and synchronization have been examined in several previous researches. Unlike our work, which is intended to support general FPGA programming, all previous works of which we are aware target coherence and synchronization among soft cores [11] [12] implemented on the FPGA. Most implementations of coherent soft processors make use of snoopy protocols, though a directory-based protocol has been examined [13]. In contrast to our scalable, network-based implementation, all of these implementations appear to rely on crossbars to communicate between caches in relatively small, single coherence domains. These structures are neither scalable nor can they be easily partitioned among multiple FPGAs. Moreover, since existing coherence work targets multi-core processors, existing coherence implementations assume fixed coherence domains among the processors, rather than granting the programmer freedom to describe free-form coherence domains. Although we have not yet attempted to use our work in the context of soft processors, we believe that our cache interface could be used to support a distributed soft multi-core architecture using the `testAndSet` atomic operation.

Like coherence, most efforts at implementing synchronization in FPGAs have occurred in the context of soft-cores, either to provide synchronization mechanisms among the cores [10] [14] or between the cores and hardware

TABLE III: FPGA RESOURCE UTILIZATION AND MAXIMUM FREQUENCY FOR OUR PRIMITIVES

(a) VARIOUS PRIMITIVES*

Primitive	Slice Registers	Slice LUTS	18K-bit BRAM	P&R f_{max} (MHz)
Lock Master	122	202	0	255
Lock Slave	81	176	0	383
Barrier Master	122	185	0	333
Barrier Slave	77	129	0	491
CS Controller	3721	5252	9	112
CS Client	2985	5721	7	113
PS Client	1660	2010	4	162

* Memory primitives target 64-bit data words with a 14-bit word address (an 8KB cache of a 128KB memory space). For comparison, the area of a private LEAP Scratchpad addressing a similar region is also shown.

(b) VARIOUS CACHE PARAMETERS**

Parameter	Slice Registers	Slice LUTS	18K-bit BRAM	P&R f_{max}
512-entry Cache	1.00	0.98	0.86	1.05
2048-entry Cache	1.00	1.11	1.57	1.11
4096-entry Cache	1.00	1.17	2.71	1.03
13-bit Addressing	1.00	0.99	1.00	1.19
15-bit Addressing	1.01	0.97	1.14	1.06
32-bit Data	1.00	1.00	1.00	1.00
128-bit Data***	1.41	1.55	1.71	1.11

** Results are normalized to the CS client baseline in Table IIIa.
*** The CS client's cache line size is changed to 128-bit.

accelerators [15]. Some of these implementations are bus-based [16] [17] [15], limiting scalability within an FPGA and preventing expansion to multiple FPGAs. Since the existing implementations are processor-specialized, their interfaces typically involve memory addresses which are used to denote mutexes. As a result, these implementations may require more area to implement than our logic-integrated locks (as much as 3x in the case of [10].) Existing implementations provide only lock/mutex management, mostly for the implementation of soft-processor atomic operations. Barriers must be implemented in software on top of mutexes, which may limit performance.

A second set of work focuses on building a cache coherent interface between an accelerator FPGA and host processors. The most recent of these works is a product from Intel, the QPI-based QuickAssist interface [18] [19]. QuickAssist manages both a coherence interface to an attached processor and the FPGA-local DRAM in-fabric, presenting a single processor-coherent memory interface to the FPGA. Unlike our memory interface, which is designed to allow multiple distributed memory interfaces, QuickAssist supports only a single coherence interface and does not admit of further scaling, due to the difficulty of meeting coherence-protocol-level timing in the FPGA. If these timing issues could be resolved, our coherence interface could be used to bridge the QuickAssist infrastructure and multiple FPGAs, providing processor-FPGA coherence across a network of FPGAs.

VII. CONCLUSION

FPGAs have become interesting computational platforms. However, the infrastructure for programming FPGAs is still lacking, especially in the realm of shared memory parallel programing. In this paper, we provide memory coherency and synchronization primitives to support designing parallel algorithms on FPGAs. We propose the coherent scratchpad, which provides a simple SRAM-like interface and automatically manages coherent caches. We also provide lock and barrier primitives, which leverage the native communications primitives of the FPGA rather than relying on shared memory. Because our primitives are framed in terms of high-level latency-insensitive channels, they may be automatically partitioned across any configuration of FPGAs that the programmer requires.

This paper describes our coherent memory interface and the first implementation of coherent scratchpads. Because of this abstraction, other coherence protocols and further performance optimizations can be applied in the future and will be transparent to the programs using coherent scratchpads.

REFERENCES

[1] H. K.-H. So and R. Brodersen, "Improving usability of FPGA-based reconfigurable computers through operating system support," in *FPL*, 2006.

[2] K. Fleming, M. Adler, M. Pellauer, A. Parashar, Arvind, and J. S. Emer, "Leveraging latency-insensitivity to ease multiple FPGA design," in *FPGA*, 2012.

[3] M. Adler, K. Fleming, A. Parashar, M. Pellauer, and J. S. Emer, "LEAP Scratchpads: Automatic memory and cache management for reconfigurable logic," in *FPGA*, 2011.

[4] C. He, W. Zhao, and M. Lu, "Time domain numerical simulation for transient waves on reconfigurable coprocessor platform," in *FCCM*, 2005.

[5] M. Shafiq *et al.*, "Exploiting memory customization in FPGA for 3D stencil computations," in *FPT*, 2009.

[6] L. A. Barroso and M. Dubois, "Cache coherence on a slotted ring," in *ICPP*, 1991.

[7] B. H. A. Ahmed, P. Conway and F. Weber, "AMD Opteron shared memory MP systems," in *Hot Chips*, 2002.

[8] M. R. Marty and M. D. Hill, "Coherence ordering for ring-based chip multiprocessors," in *MICRO*, 2006.

[9] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[10] V. Mirian and P. Chow, "Managing mutex variables in a cache-coherent shared-memory system for FPGAs," in *FPT*, 2012.

[11] H. Lange, T. Wink, and A. Koch, "MARC II: A parametrized speculative multi-ported memory subsystem for reconfigurable computers," in *DATE*, 2011.

[12] V. Mirian and P. Chow, "FCache: A system for cache coherent processing on FPGAs," in *FPGA*, 2012.

[13] ——, "An implementation of a directory protocol for a cache coherent system on FPGAs," in *ReConFig*, 2012.

[14] E. Matthews, L. Shannon, and A. Fedorova, "Polyblaze: From one to many bringing the Microblaze into the multicore era with Linux SMP support," in *FPL*, 2012.

[15] D. Andrews *et al.*, "Achieving programming model abstractions for reconfigurable computing," *IEEE Trans. on VLSI*, vol. 16, no. 1, 2008.

[16] Xilinx, Inc., "LogicCore IP Mutex," 2010. [Online]. Available: http://www.xilinx.com/

[17] Altera Corporation, "Mutex Core," 2009. [Online]. Available: http://www.altera.com/

[18] L. Ling *et al.*, "High-performance, energy-efficient platforms using in-socket FPGA accelerators," in *FPGA*, 2009.

[19] N. Oliver, "A reconfigurable computing system based on a cache-coherent fabric," in *CARL: Intersections of Computer Architecture and Reconfigurable Logic*, 2012.