On-chip Communication and Synchronization Mechanisms with Cache-Integrated Network Interfaces

Stamatis Kavadias, Manolis Katevenis
Michail Zampetakis, Dimitrios S. Nikolopoulos*

Institute of Computer Science
Foundation for Research and Technology - Hellas (FORTH)
Heraklion, Crete, Greece - member of HiPEAC
{kavadias,kateveni,mzampet,dsn}@ics.forth.gr

ABSTRACT

Per-core local (scratchpad) memories allow direct inter-core communication, with latency and energy advantages over coherent cache-based communication, especially as CMP architectures become more distributed. We have designed cache-integrated network interfaces (NIs), appropriate for scalable multicores, that combine the best of two worlds –the flexibility of caches and the efficiency of scratchpad memories: on-chip SRAM is configurably shared among caching, scratchpad, and virtualized NI functions. This paper presents our architecture, which provides local and remote scratchpad access, to either individual words or multiword blocks through RDMA copy. Furthermore, we introduce *event responses*, as a mechanism for software configurable synchronization primitives. We present three event response mechanisms that expose NI functionality to software, for multiword transfer initiation, memory barriers for explicitly-selected accesses of arbitrary size, and multi-party synchronization queues. We implemented these mechanisms in a four-core FPGA prototype, and evaluated the on-chip communication performance on the prototype as well as on a CMP simulator with up to 128 cores. We demonstrate efficient synchronization, low-overhead communication, and amortized-overhead bulk transfers, which allow parallelization gains for fine-grain tasks, and efficient exploitation of the hardware bandwidth.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: MEMORY STRUCTURES—*Design Styles*[Cache memories; Shared memory; Virtual memory]; B.4.1 [**Hardware**]: INPUT/OUTPUT AND DATA COMMUNICATIONS—*Data Communications Devices*; B.6.1 [**Hardware**]: LOGIC DESIGN—*Design Styles*[Memory control and access]

*The authors are also with the University of Crete, Greece

CF'10, May 17–19, 2010, Bertinoro, Italy.

General Terms

Design, Performance

Keywords

network interface, explicit communication, synchronization, cache

1. INTRODUCTION

As the number of processing cores per chip increases, so does the need for efficient and high-speed communication and synchronization support, so that applications can exploit the numerous available cores. In small systems, communication and synchronization is often done *implicitly*, through coherent caches, complemented by hardware prefetching for performance reasons. Although the use of caches relieves software from locality and communication management, as CMP architectures will become more distributed, indirection through directories and the round-trip nature of coherence will introduce larger communication latency.

Recently, researchers and implementors have begun to reassess the use of directly-addressable, per core local memories for a wide range of target application domains. Scratchpad memories (also termed "Local Stores" and "Stream Register Files" in the literature) lend themselves naturally to the use of *remote DMA (RDMA)* transfer-offload engines and allow direct, software controlled scratchpad-to-scratchpad communication. Direct transfers are expected to provide both performance and energy advantages over coherence. In addition, the combination of hardware assisted transfers with per core local memories, enables the use of producer-initiated communication as well as overlapping of communication with computation even using simple processors. To exploit these opportunities though, software needs to engage in locality and communication management.

In this paper, we present the architecture of cache integrated network interfaces (NI) that we developed in the context of the "SARC" project [1], to achieve the best of two worlds: the flexibility of caches and the efficiency of scratchpad memories. The proposed architecture allows sharing on-chip SRAM for caching, scratchpad and NI communication functions at cache-line granularity, all mapped in the application's virtual address space for virtualization. We support load/store access to remote scratchpads, and RDMA's that can be *explicitly acknowledged*. These communication mech-

anisms use virtual source and destination addresses and thus provide generalized read and write *accesses* for explicit communication.

In addition, we introduce event-responses as a framework for hardware synchronization mechanisms configurable by software. Event responses leverage line tag and state lookup in the normal cache access flow to expose NI functionality to software. Three event-response mechanisms are discussed: *command buffers* used to initiate multi-word communication; *counters* that provide barriers for software selected sets of arbitrary size accesses; and *multiple-reader queues* implemented in scratchpad memory for multi-party synchronization.

We have implemented the above framework in a 4-core FPGA-based hardware prototype. Our evaluation on the prototype provides a comparison of RDMA-copy and remote store mechanisms for on-chip communication. Remote stores with write combining provide low-overhead communication for short data transfers and provide gains from the parallelization of fine-grain tasks of less than 500 processor cycles length. RDMA-copy transfers are more bandwidth efficient and amortize software initiation and communication overhead. The presented communication mechanisms enable also maximal utilization of the available on-chip and off-chip memory bandwidth with buffering space as little as 3 KB per scratchpad memory. Simulation of barriers and contended locks, based on counters and multiple-reader queues, for up to 128 cores, shows 3× to 5× improvement over tree-based barriers and MCS locks [21] that use coherently cached variables.

The rest of this paper is organized as follows: In section 2 we review related work. Section 3 presents our architecture in detail. In section 4, we briefly describe the hardware and software platforms of our study. Section 5 presents performance evaluation on the FPGA hardware prototype and with simulations, and section 6 concludes the paper.

2. RELATED WORK & CONTRIBUTIONS

Prior work on fine-grain access control [27] and application specific coherence protocols [5] demonstrates how lookup mechanisms leverage local or remote handling of coherence events and has influenced our approach to cache-integration of event responses. The Cray T3E [28] supported single-reader queues and barrier/eureka FSM-based synchronization hardware. Our counter-based barrier support is more general and easier to virtualize. The MIT Multi-ALU Processor (MAP) [14], provided support for direct message transmission/reception from registers and dedicated receive side buffering. Our design avoids communication arrival interrupts required in MAP, exploiting virtual memory destinations and explicit acknowledgments in scratchpad memory.

Ranganathan et al. [24] propose associativity-based partitioning and overlapped wide-tag partitioning of caches for software-managed partitions (among other uses). Associativity-based partitioning allows independent, per way addressing, while overlapped wide-tag partitioning adds configurable associativity. PowerPCs allow locking caches (misses do not allocate a line) (e.g. [11]). Intel's Xscale microarchitecture allows per line locking for virtual address regions either backed by main memory or not [12]. Our design generalizes the use of line state for configurable communication initiation (subsection 3.3) in addition to locking lines in the cache.

Syncretic adaptive memory (SAM) [29] integrates a stream register file (SRF) with a cache and uses cache tags to identify segments of generalized streams. It also integrates a compiler managed translation mechanism to map program stream addresses to cache and main memory locations. Compared to our architecture, SAM requires a specialized compiler to exploit integration of cache with a SRF and does not provide event response support.

In smart memories [6] the first level of the hierarchy is reconfigurable and composed of 'mats' of memory blocks and programmable control logic. This enables several memory organizations ranging from caches that may support coherence or transactions, to streaming register files and scratchpads. Their design exploits the throughput targeted processor tiles to hide increased latencies because of reconfigurability. Smart memories incur significant area overhead which we estimate to be higher than our integrated approach[1]. It should be possible to support coherent cache and scratchpad organizations simultaneously and program smart memories for event responses with adequate microcode memory, though the authors do not consider these techniques. SiCortex [25] ICE9 chip features microcode-programmable tasks in a coherent DMA engine side-by-side with an L2 cache shared by 6 cores, but does not support scratchpad memory. Smart memories and ICE9 DMA engine are the most similar to our cache-integrated NI, but our work focuses on keeping the NI simple enough to integrate with a high performance cache.

Leverich et al. [17] provide a detailed comparison of caching-only versus partitioned cache-scratchpad on-chip memory systems for CMPs. They find that hardware prefetching and non-allocating store optimizations in the caching-only system eliminate any advantages in the mixed environment. We believe their results are due to considering communication between on-chip cores and off-chip main memory. By contrast, for on-chip core-to-core communication, RDMA provides significant traffic reduction, which together with event responses and NI cache integration are the focus of our work. Streaming hardware support for general purpose systems exploiting caches for streaming data was considered in [10, 9, 23, 4]. In [10] cache control bits are used for best-effort avoidance of replacements and scatter-gather enhancements of the L2 controller are proposed for a single-core system. Streamware [9] exploits the compiler to avoid replacements of streaming data mapped to processor caches, for codes amenable to stream processing. Rangan et al. in [23], study alternatives to hardware support for pipelined streaming. They conclude that optimizations for write-forwarding [16, 22, 2, 15] at line boundaries, synchronization counters in L2 caches (which they do not describe) and dedicated receive-side caches in a separate address space for pipelined streaming data, can reach the performance of heavyweight hardware support. Our design integrates equivalent mechanisms inside caches augmented with RDMA for efficient bulk transfers.

3. CACHE-INTEGRATED NETWORK INTERFACE MECHANISMS

Explicit communication and synchronization mechanisms

[1]A direct comparison requires porting our FPGA design to an ASIC flow, because the work on smart memories only provides estimates of silicon area for an ASIC process.

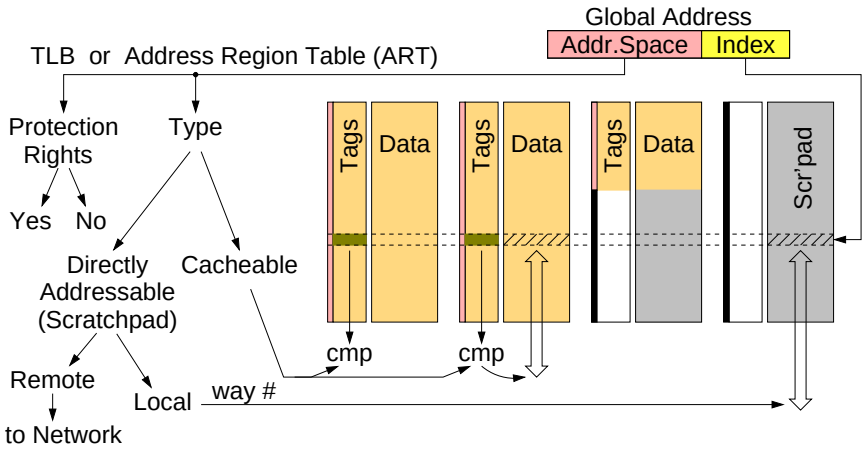

Figure 1: Memory access flow: the tags of the (configurable) scratchpad areas are not used

work like network I/O devices: the processor initiates operations, polls for status, or waits for input or notifications using *memory-mapped* control and status registers. To increase parallelism, multiple pending operations must be supported, hence there must exist multiple control and status registers. To reduce overhead, these multiple registers must be *virtualized*, so as to be accessible in user-mode. To reduce latency, these mechanisms and registers need to be brought close to the processor, at the level of cache memory, as opposed to the level of main memory or I/O bus. This section explains how we achieve all of the above, describing our communication (RDMA) and synchronization (counters, queues, notifications) mechanisms and some typical uses. Although we chose to implement our mechanisms at the level of *private L2 caches*, the ideas are general and independent of that choice. We integrated our NI mechanisms into *private* –as opposed to shared– caches in order for processors to have parallel access to them. And we integrated them into *L2* caches –as opposed to L1 caches or processor registers– in order to provide sufficient scratchpad space for application data and sufficient number of time-overlapped communication operations, and in order not to affect the processor clock. Our prototype implements a phased, pipelined L2 cache (1 access per cycle), a write-through L1 cache, and selective L1-caching of L2 scratchpad regions.

3.1 Memory Access Semantics: Cache, Scratchpad, Communication

We explained above the advantages possible by scratchpads and multiple memory-mapped communication control/status "registers", all brought close to the processor into private caches. To support these, memory *access semantics* must vary. We use two mechanisms to signal such modified semantics: *address translation*, to mark entire address regions as explicitly managed or scratchpad (figure 1), and cache *line state* bits, to indicate different access semantics (figure 2).

As shown in figure 1, we assume that the address translation mechanism is augmented with a few bits that mark, for each address region or page, whether it contains cacheable or directly-addressed (scratchpad) data. This is important for *remote* scratchpad regions, because it indicates to hardware which accesses are remote and thus do not request

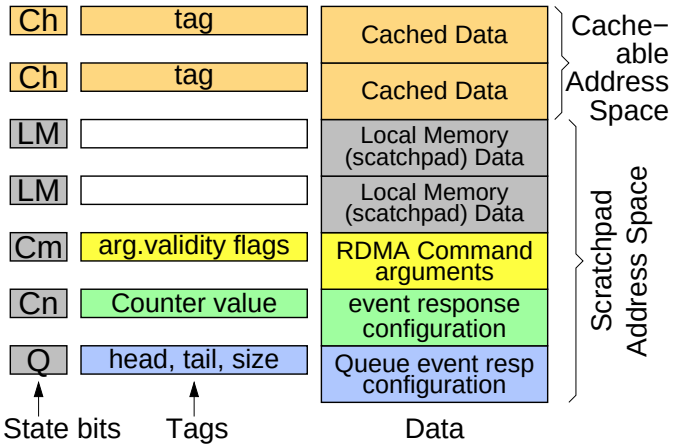

Figure 2: Cache line types: state bits mark lines with communication semantics

local caching and the relevant space allocation; note that *remote stores* with *write-combining buffer(s)* is a very efficient method for producers to send data to consumers (section 5.2). *Local* scratchpad regions, on the other hand, are marked both through the address translation path as above, and using the state bits of cache lines to mark as *non-evictable*. The former obviates tag bit comparison to verify that a memory access actually hits into a scratchpad line[2]; in this way, tag bits of scratchpad areas are freed, and we use them for other purposes in the case of communication semantics. Scratchpad lines must still be marked through their state bits, so that hit/miss searches in cacheable space will ignore their tag bits. This combined mechanism allows for *runtime-configurable partitioning* of the on-chip SRAM blocks between cache and scratchpad use, thus adapting to the needs of the application that is being run at each point in time.

The multiple, *virtualized* communication *control/status*

[2]in our SARC project, we use *Address Region Tables (ART)*, instead of the traditional TLB's, in order to support scalable page migration (only local ART's are updated on local migrations –not all TLB's throughout the entire system), as explained in [13]; that issue, however, is orthogonal to what we discuss in this paper.

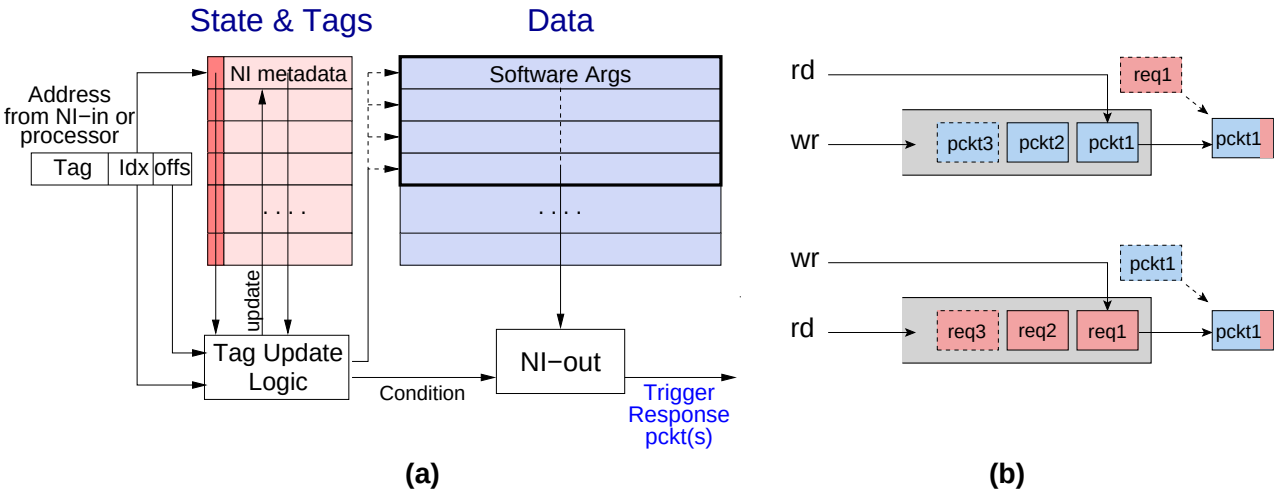

Figure 3: (a) Event Response mechanism integration in the normal cache access flow. (b) Multiple-reader Queue function illustration.

"registers" that we desire have to be implemented in explicitly managed address regions to indicate their semantics via state and tag bits. Other than plain scratchpad memory, cache lines in scratchpad space can be marked, in their state bits, as having *three* kinds of such special semantics, as shown in figure 2: *(i)* communication *(RDMA)* command/status; *(ii) counter*, used for synchronization and notification through atomic increment operations; or *(iii) queue descriptor*, used again to atomically multiplex or dispatch information from/to multiple asynchronously executing tasks. The semantics and use of these primitives, including the use that they make of the tag bits, will be explained in the rest of section 3. Their *virtualized* nature results as follows. Since they are located in memory space, processors can only access them going through address translation and protection. Also, as many communication "registers" as desired can be freely allocated in the (virtual) address space of any process. Thus, each process can freely access, in user-mode, its own special "registers", independent of and asynchronously to other processes. Virtualization also requires that address arguments passed to control registers are given in *virtual* –rather than physical– space, and are protection-checked by hardware; we assume that the network interface has access to a second port of the TLB (or ART) in order to perform these. When the operating system swaps a scratchpad out of a certain cache, it has to properly mark and record these special cache lines; to do so, it has to either know their address and type beforehand, or else it can discover them by reading the state and tag bits, which is feasible in our architecture because these bits are mapped to a special address range.

3.2 RDMA Communication: in-Place Data Delivery

Remote direct memory access (RDMA) is widely used as the basic and most efficient primitive for explicit communication, especially for large volumes of data. Relative to the delivery of data into receive queues, it has the advantage of *zero-copy*, by directly delivering "in-place". Compared to the copying of data via load-store instructions, it has the advantage of *asynchrony*, thus allowing communication to overlap with computation. Unlike implicit communication through cache coherence, it can deliver data to the receiver *before* the receiver asks for them, thus eliminating read-miss latency. Finaly, relative to the cases of successful prefetching in caches, RDMA uses much fewer packets to perform the transfer, thus economizing on energy. Large RDMA transfers are broken by the hardware into multiple smaller packets. Even if these packets follow different routes through the network –since *adaptive routing* greatly improves network performance– packets arriving out-of-order are correctly assembled in-place, since each carries its own destination address; RDMA completion detection becomes harder, and we handle it as discussed in section 3.4.

In order to initiate an RDMA operation, software must pass 4 arguments to the hardware: size, source, destination, and acknowledgment addresses. We assume that software writes each of these 4 values exactly *once* into a *command buffer* specially marked in scratchpad space, as shown in figure 2. These 4 writes can occur in any order, separated by any time distance; the hardware monitors them using *validity flag* bits, kept in the (otherwise unused) tag bits; when all 4 writes have occurred, the RDMA is triggered. This mechanism works correctly, in user-mode, independent of how many threads or tasks do the same thing in parallel in separate command buffers, and even on processors where store instructions may complete out-of-order[3]. Remote DMA operations may occur between scratchpad regions, or between scratchpads and non-cacheable portions of main memory; strided RDMA and RDMA to/from cacheable addresses are on our future-extensions list. When very short blocks of data are to be transferred, RDMA initiation, by storing 4 arguments per transfer, incurs non-negligible overhead; in those cases, *remote stores* (regular store instructions, to remote scratchpad addresses) are more efficient –especially using write-combining buffer(s) (section 5.2).

3.3 Event Responses

Event responses provide a framework for hardware syn-

[3]with out-of-order store's, pushing the 4 arguments, in-order, into a FIFO structure at a *single* memory address would *not* work

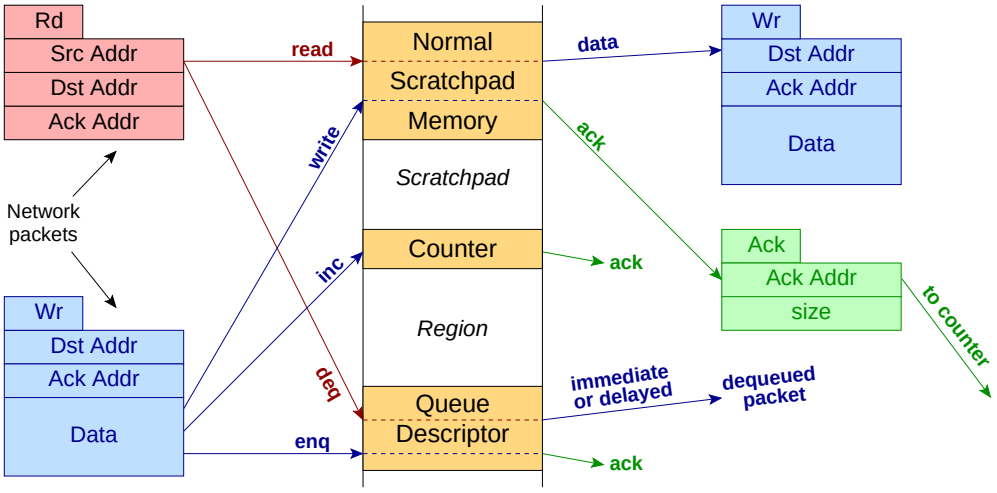

Figure 4: Remote access to scratchpad regions and generation of explicit acknowledgements.

chronization mechanisms configurable by software. Event-response mechanisms are supported by event sensitive lines (ESLs) with dedicated states as shown in figure 2. On every access to the cache, local or remote, normal cache operation checks the state and tag bits of the addressed line(s). The NI monitors all accesses to ESL's, called *events*, reads and updates state stored in the ESL tag, and checks whether some conditions are met, as illustrated in figure 3(a). When the the relevant condition is fulfiled *software-preconfigured* communication activity is triggered, which we term the *response*. Software configuration is indicated by communication arguments are stored in the data block of the ESL that are used by the outgoing network interface (NI out).

We designed three event-response mechanisms, for *command buffers*, *counters* and *multiple-reader queues*. Command buffers are used to send messages or initiate RDMA-copy operations, as discussed in section 3.2. Counters are intended to provide software notification regarding the completion of an unordered sequence of operations (e.g. multiple transfer reception, or arrivals at a barrier). Counters support atomic add-on-store and up to four configurable notification addresses. When the counter reaches zero, a preconfigured word is sent to the four addresses. Figure 3(b) shows how a multiple-reader queue (mr-Q) works. The mr-Q allows dequeue (read) operations to wait until data arrive at the queue, effectively *matching* read and write requests in time. When a write is matched with a read in the mr-Q, a response packet is triggered, with the data of the write sent to the response address of the read. Reads and writes to the mr-Q are buffered in scratchpad memory contiguous to the ESL, forming the queue body. Head and tail pointers stored in the ESL tag are used to modify some of the least significant index bits to access a queue entry in the data array, as shown in figure 3(a).

3.4 Software Notification via Explicit Acknowledgements

To allow the enforcement of a software desired order among read or write accesses to remote scratchpads, or synchronize computation with such accesses, all explicit transfers can be acknowledged. Explicit acknowledgements can be accumulated in counters for completion notifications of one or more multiword transfers, *even over an unordered network*.

Figure 4 shows how acknowledgements are generated for each NoC packet. Three types of lines inside a scratchpad region are depicted in the middle, with read and write request packets arriving from the left, and the corresponding generated reply packets on the right. A read packet arriving to normal scratchpad memory generates write reply packets according to the destination and acknowledgement addresses in the read (this is also true for counters –not shown). When a read arrives at a queue, the write reply may be delayed. Writes arriving at any type of line generate acknowledgements toward the acknowledgement address in the write and the size of the write packet as the data. This size can be accumulated in counters for completion notification of the initial transfer request (read or write). Acknowledgements arriving at any type of line act as writes (not shown), but do not generate further acknowledgements.

4. HARDWARE AND SOFTWARE PLATFORMS

We have fully implemented the architecture described in section 3 in a hardware prototype based on Xilinx Virtex-5 FPGA. A previous version of the prototype was presented in [7]. The current version is a major rewrite of the code, optimized for logic reuse, implementing event responses, three levels of NoC priority and some other features not present in the version of [7]. In addition, in this paper, we model event responses in GEMS [19] and simulate counter-based barriers and mr-Q based locks for up to 128 processors.

The block diagram of the FPGA system is presented in figure 5. There are four Xilinx microblaze IP cores, each with 4KB L1 instruction and data caches and a 64KB L2 data cache where our network interface mechanisms are integrated. An on-chip crossbar connects the 4 processors through their L2 caches and NI's, the DRAM controller –to which we added a DMA engine– and the interface (L2 NI) to a future second-level, 3-plane 16×16 interconnect, over 3 high speed RocketIO transceivers, to make a 64-processor system. All processors are directly connected over a bus

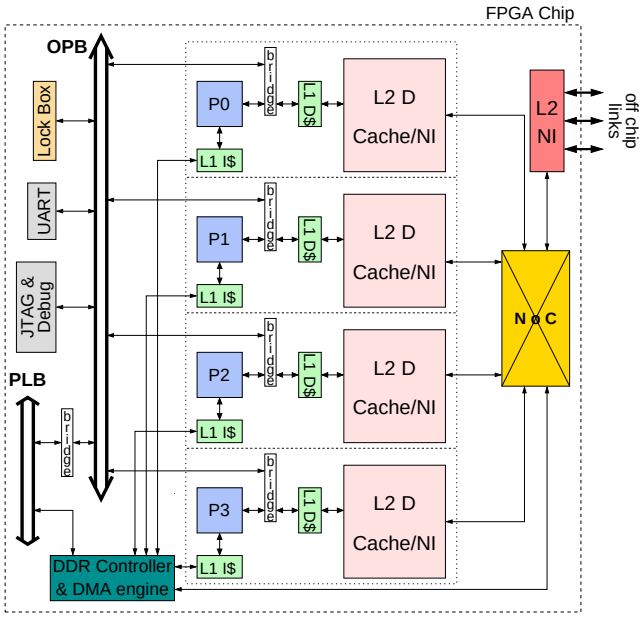

Figure 5: FPGA prototype system block diagram.

(OPB) to a hardware lock-box supporting a limited number of locks, used for comparison purposes (section 5.2).

4.1 Prototype Software Environment

For our hardware and software synthesis we used the ISE design suite and the Embedded Development Kit (EDK) tools, which provide a complete flow for RTL-based designs and Intellectual Property (IP) components. For compiling software, we used a version of gcc, *mb-gcc*, targeted to Microblaze processors. For debugging, we used the Xilinx Microprocessor Debug (XMD) engine, which can be used while a microprocessor is running on the system.

We implemented the *sylib, scrlib* and *nilib* libraries for parallelization, synchronization, and communication. The *syslib* library implements locks, barriers, memory allocation, and basic timing and I/O facilities; it provides alternative implementations of locks and barriers, thread-safe memory allocation, thread-safe I/O functions, and basic mechanisms for getting a core ID and the value of a global system timer. The *scrlib* library manipulates scratchpad memory: allocate a part of the L2 cache memory as scratchpad space at runtime, convert local addresses to remote addresses, and check if an address is local or remote. The library also includes mechanisms for marking a cache line as a queue, a register, or a control line. Last, *nilib* contains functions for preparing and issuing DMAs, for managing command buffers, notifications, and queues, and for sending messages to remote scratchpad memories.

4.2 Simulation Infrastructure

Our simulations are based on the GEMS memory system simulator, driven by accesses from the Simics [18] full system simulation. GEMS supports defining cache states, state transitions, and associated actions; thus, we directly model event response mechanisms. A similar set of libraries to those of subsection 4.1 where developed to run over Simics. For our measurements we use the Simics support for light weight instrumentation, using simulation break instructions, to selectively measure synchronization primitive invocation intervals excluding the surrounding loop code.

5. PERFORMANCE EVALUATION

The objective of our evaluation is to demonstrate that the proposed architecture achieves low core-to-core communication latency, low latency for high-level synchronization primitives –locks and barriers–, effective utilization of the available on-chip and off-chip memory bandwidth and exploitation of fine-grain parallelism in applications. To this end, we evaluate a FPGA prototype of the proposed architecture with controlled microbenchmarks and applications, both stressing on-chip communication and simulate the performance of locks and barriers for large core numbers.

5.1 Applications and Benchmarks

The STREAM triad benchmark [20] is designed to stress bandwidth at different layers of the memory hierarchy. The benchmark copies three arrays from a "remote" to a "local" memory, conducts a simple calculation on the array elements and sends the results back to original "remote" memory. We developed two configurations of STREAM for stressing on-chip and off-chip memory bandwidth respectively. In the on-chip configuration, the data is streamed from scratchpad memories to scratchpad memories and backwards, whereas in the off-chip configuration, data is streamed from DRAM to scratchpad memories. In each configuration, we apply multi-buffering to overlap the latency of fetching data from the "remote" memory and we vary the number of buffers, the buffer size and the communication mechanism, which alternates between DMAs and remote stores.

The FFT benchmark originates from the StreamIt language benchmarks [26] and includes all-to-all data exchange patterns between processors. We configure the benchmark so that it performs the entire computation and all-to-all data exchanges on-chip, in order to stress the performance of our cache-integrated NI mechanisms. We implement data exchanges using DMAs and remote stores to explore trade-off's between the two communication mechanisms.

The bitonic sort benchmark originates also from the StreamIt language benchmarks [26]. Bitonic-sort is computation bound and we use it to measure the minimum granularity of exploitable parallelism on the architecture. We configure the benchmark so that sorting and any associated data exchanges between processors are performed entirely on-chip and we explore the trade-off between DMAs and remote stores in the implementation of the benchmark.

The simulated microbenchmarks compare coherence-based implementations of tree-based barriers and MCS locks described in [21], with barriers composed from counters and mr-Q based locks respectively. Counter barriers are realized by constructing an arrival and a broadcast tree of counters. All threads store the value one (1) to counters on the arrival tree leaves and then poll on local flags for the barrier completion. The counters count arrivals and forward notifications of completion of the barrier phases. The mr-Q implements a lock as follows: initially we store a token in it; then, waiting for a read from the mr-Q acts as lock acquisition and writing back the token acts as lock release. For barriers, we measure 10^4 episode times independently on each processor and average cycle times across processors and iterations. For locks, we measure 1-8 thousand lock acquisitions and releases per core, with an empty critical section.

(a) Contended lock times

	Mutex Lock			MRQ Lock		
CPUs	1	2	4	1	2	4
Acquire	44	87	201	42	82	193
Release	32	32	32	21	21	21
Total	76	119	223	63	103	214

(b) Barrier times

	Mutex Barrier			Counter Barrier		
CPUs	1	2	4	1	2	4
Cycles	188	281	618	75	105	117

Table 1: Different lock and barrier latencies.

5.2 Results

5.2.1 Performance on the Hardware Prototype

We implemented two versions of locks and barriers on our hardware prototype. The mutex lock uses the hardware lock box of figure 5, whereas the second uses multiple reader queues (see Section 3). For barrier synchronization, we developed a barrier using the mutex lock implemented with the hardware lock box and a barrier using counters.

Table 1(a) illustrates that the lock that leverages the multiple reader queue executes an acquire-release pair for an empty critical section under full contention between 4 cores in 214 cycles. Table 1(b) illustrates that a simple counter-based barrier with on-chip communication of the barrier arrival and release notification between 4 cores takes 117 cycles. To put these numbers in perspective, we note that the one-way latency of a remote store is 35 cycles. Both the lock and the barrier require one remote store each. The barrier uses a remote store to a counter from which an automatic notification is generated when the counter value reaches 0. Software overhead accounts for 28 cycles in the case of locks and 34 cycles in the case of barriers. Though direct comparisons with competitive hardware designs are not possible on our FPGA prototype, we note that on leading commercial multicore processors, lock acquire-release pairs cost in the order of thousands of cycles (typically around 1 ns on processors with multi-GHz clocks), and barriers cost in the order of tens of thousands of cycles (typically over 5ns on processors with multi-GHz clocks) [3].

Figure 6 illustrates the results of the STREAM benchmark on our FPGA prototype. We plot the maximum feasible bandwidth on the prototype (horizontal line) for remote stores and DMAs and the realizable bandwidth while we vary the buffer size and the number of buffers used for overlapping computation with memory latency. For measuring the on-chip realizable bandwidth we lay out the data in the scratchpad memories of one or two cores and have the remaining cores stream data from these scratchpads. Off-chip bandwidth is measured by streaming data from three large arrays in DRAM. As expected, the achievable maximum bandwidth with remote stores is lower (about 7×–8×) than the maximum achievable bandwidth with DMAs, since remote stores incur the overhead of one instruction per word transferred whereas DMAs can transfer up to 64 KB worth of data with overhead of 4 instructions. The software saturates the off-chip memory bandwidth when all four cores stream data and use three or more buffers of size 1 KB for latency overlap. On-chip memory bandwidth is saturated when three cores stream data out of and in to the scratchpad of the remaining core, using three or more buffers of size 1 KB for latency overlap. In all cases, the architecture can maximize bandwidth and overlap memory latency using a small space (3KB–4KB) for buffering data in scratchpad memory.

Figure 7 illustrates the speedup of on-chip bitonic sorting for various input sizes, using remote stores and DMAs for inter-core communication, as well as the breakdown of execution time in computation and communication for selected input sizes. The results indicate two trends: First, that the proposed cache-integrated on-chip communication mechanisms enable profitable parallelization (i.e. speedup greater than 1 on 2 or more cores) of tasks as fine as 470 clock cycles (input size $N = 4$). Second, the results exhibit a trade-off between DMA-based and remote-store-based communication. In small input sizes ($N = 4 \ldots 64$), communication via remote stores is 5%–41% less than communication with DMAs since for very short (word-size) transfers, remote stores have less overhead per transfer (one vs. four instructions). With the same small input sizes, overall performance with remote stores exceeds performance with DMAs by 0.2%–14%. For larger input sizes, communication time with DMAs is 13%–32% less than communication time with remote stores, however overall performance with DMAs exceeds only marginally performance with remote stores (by no more than 0.2%) due to the low communication to computation ration of the benchmark. Parallel efficiency with DMAs and remote stores (defined as the ratio of speedup to the number of cores) reaches 89% on 2 cores and 67% on 4 cores, for data sets that fit on-chip, i.e. do not exceed the 64 KB of available on-chip L2 cache space. Overall, the presence of both communication primitives on the prototype provide the capability to parallelize effectively at both fine and coarse granularity.

Figure 8 illustrates the speedup of on-chip FFT for various input sizes, using remote stores and DMAs for inter-core communication, as well as the breakdown of execution time in computation and communication for various input sizes. The results show similar trends with bitonic sort in terms of communication performace. For small problem size ($N < 128$) remote stores accelerate communication by 5%–48% and overall performance by 0.4%–4.5%. However, FFT does not profit from parallelization on small input sizes ($N < 128$), because execution time is dominated by overhead specific to parallelization, in particular, instructions for locating the receivers of messages during the global data exchange phase of FFT and loop control overhead. For larger input sizes, DMAs outperform remote stores (by 0.6%–20% in terms of overall performance and) due to lower communication initiation overhead and better overhead amortization. The performance advantage of DMAs is consistently amplified as the input size increases. Parallel efficiency with DMAs reaches 95% on 2 cores and 81% on 4 cores, while with remote stores it is capped at 88% on 2 cores and 68% on 4 cores, for problem sizes with data sets that fit on-chip.

5.2.2 Simulated Locks and Barriers

Figure 9(a) shows the average latency of contended lock-unlock pairs of operations. In both implementations requests are queued until the lock is available. The mr-Q based implementation is about 3.6-3.9 times faster than the MCS lock implementation. This is because contentended lock-

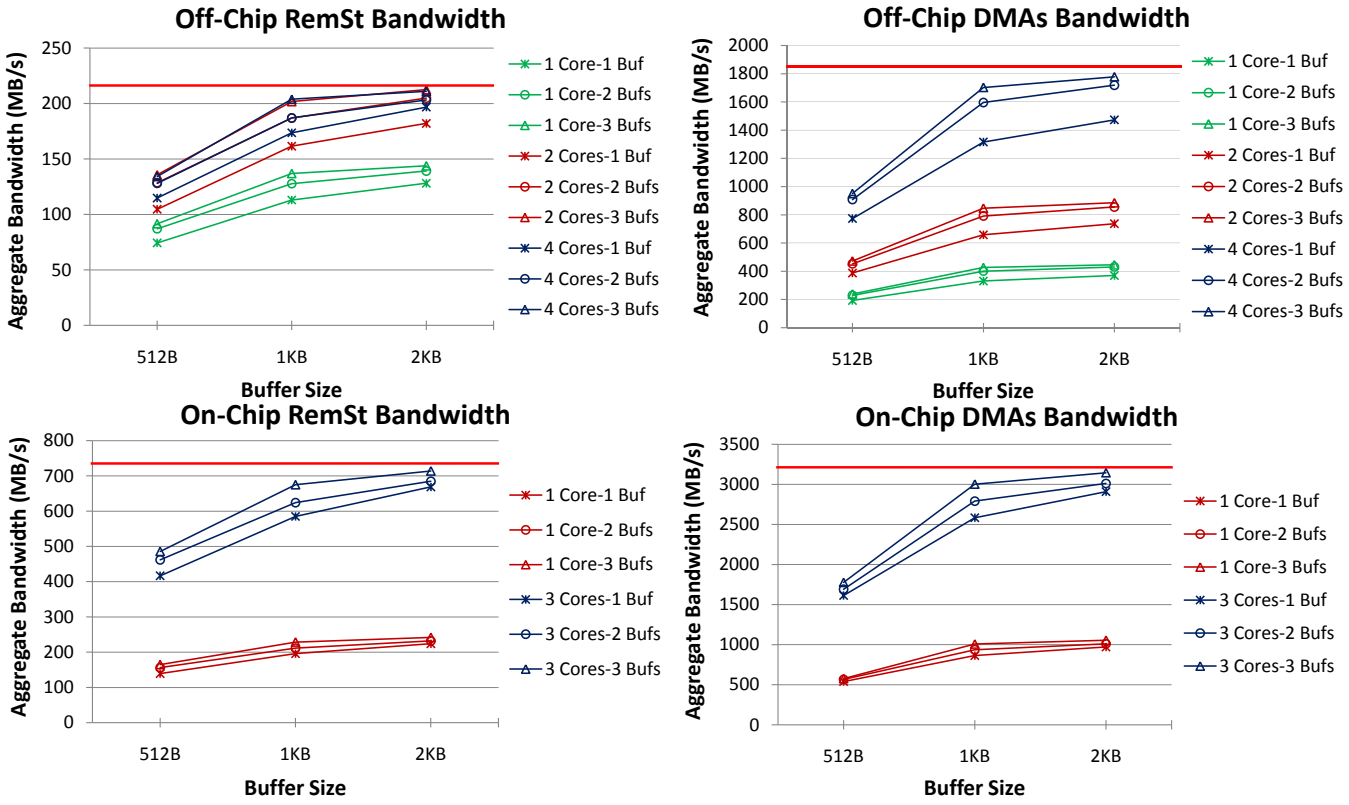

Figure 6: Performance of STREAM triad benchmark.

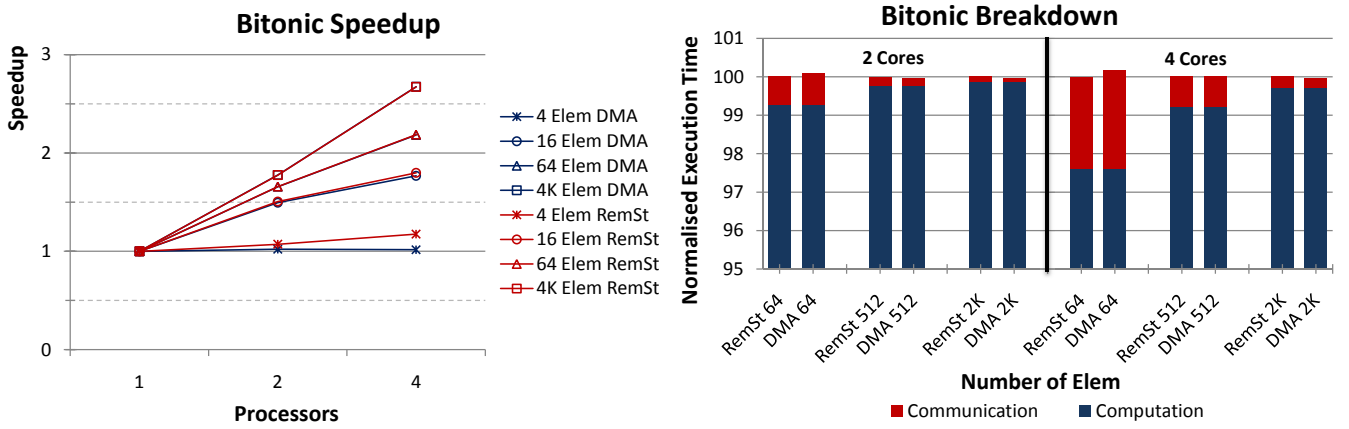

Figure 7: Performance of bitonic sort.

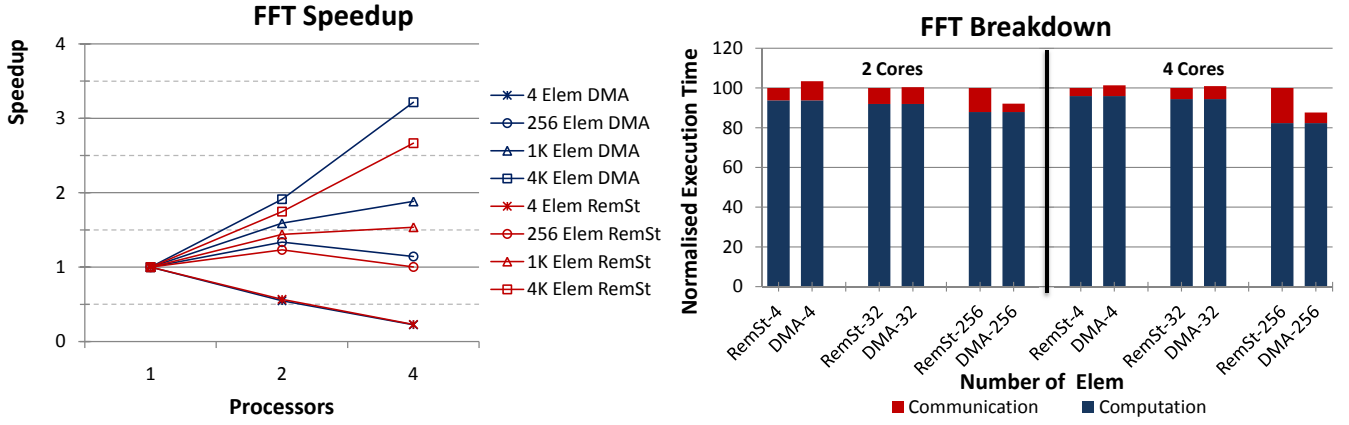

Figure 8: Performance of FFT.

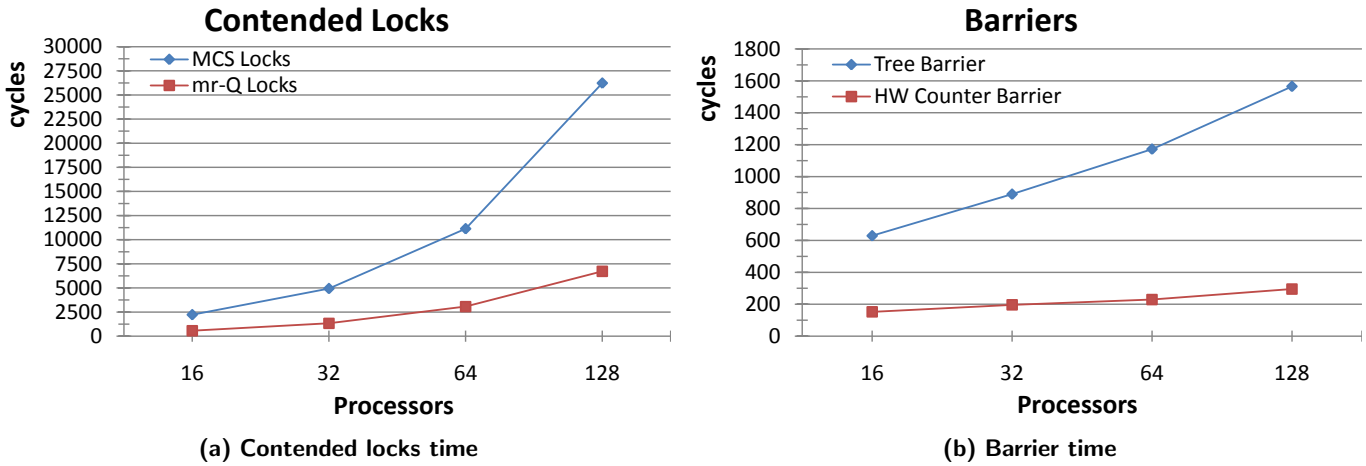

Figure 9: Average latency versus number of processors for simulated barriers and locks. MCS locks and Tree-based Barriers use coherent communication. mr-Q Locks and Counter Barriers utilize event responses and direct scratchpad-to-scratchpad communication.

unlock operations for MCS locks will incur 3 to 5 misses per iteration, requiring remote acquisition of cache lines through the directory.

Figure 9(b) shows the performance of the two barrier implementations. The counter-based barrier is from 4.1× faster for 16 cores to 5.3× faster for 128 cores. The main source of increased latency for the tree barrier using coherent variables, is that a barrier requires propagation of signals (for arrival of node or group and exit notifications). Receivers polling for these signals introduce multiple round-trips through the directory to each signal propagation.

For both MCS barriers and locks, one can expect that aggressive non-blocking coherence protocols [8] and migratory sharing optimizations can reduce the latency of contended flag update-reclaim interactions (atomic or not), but communication operations in these algorithms are dependent on each other and will introduce serialization of miss overhead. Explicit communication advocated here can significantly reduce such overheads. In addition, counters and queues can further reduce synchronization overhead by implementing the required atomicity in cache-integrated NIs and thus decoupling the processor from the synchronization operation.

6. CONCLUSIONS AND FUTURE WORK

This paper presented our design for cache-integrated network interfaces aiming to enhance the scalability and efficiency of future multicore systems. In addition, we presented our design for hardware event response mechanisms configurable by software. We evaluate this architecture on an FPGA prototype and with simulations and demonstrate its capabilities. Direct communication and event response mechanisms provide scalable synchronization to several tens of cores. Remote stores with write combining are shown that allow parallelization gains from tasks of less than 500 cycles in legth. For the benchmarks used, the communication efficiency of RDMA quickly dominates over remote stores for relatively small data transfers. Finaly, RDMA transfers can saturate the on-chip bandwidth with as few as three overlapped transfers of 1KB on our prototype.

As CMP architectures become more distributed, mechanisms for on-chip direct communication will allow performance gains from the many cores available, as well as increased hardware efficiency and software pre-configured communication via event responses will be able to support latency secitive tasks providing better scalability.

Acknowledgements

This work was supported by the European Commission in the context of the projects SARC (FP6 IP #27648) and the HiPEAC Network of Excellence (NoE 004408). We also thank, for their assistance in designing the architecture and in implementing the prototype: Vassilis Papaefstathiou, Giorgos Kalokairinos, George Nikiforos, Dionisios Pnevmatikatos, Dimitris Nikolopoulos, Alex Ramirez, Georgi Gaydadjiev, Spyros Lyberis, Christos Sotiriou, Euriclis Kounalakis, Dimitris Tsaliagos, and Michael Ligerakis.

7. REFERENCES

[1] SARC: Scalable computer ARChitecture, 2005-2009. European IP Project.

[2] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *HPCA'97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, page 204, Washington, DC, USA, 1997. IEEE Computer Society.

[3] G. Bronevetsky, J. Gyllenhaal, and B. R. de Supinski. CLOMP: Accurately Characterizing OpenMP Application Overheads. In *Proc. of the Fourth International Workshop on OpenMP (IWOMP)*, pages 13–25, West Lafayette, IN, May 2008.

[4] H. Cook, K. AsanoviÄĞ, and D. A. Patterson. Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments. Technical Report UCB/EECS-2009-131, EECS Department, University of California, Berkeley, Sep 2009.

[5] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-specific protocols for user-level shared memory. In *Supercomputing '94: Proceedings of the*

1994 conference on Supercomputing, pages 380–389, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[6] A. Firoozshahian, A. Solomatnikov, O. Shacham, Z. Asgar, S. Richardson, C. Kozyrakis, and M. Horowitz. A memory system design framework: creating smart memories. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 406–417, New York, NY, USA, 2009. ACM.

[7] George Kalokairinos, Vassilis Papaefstathiou, George Nikiforos, Stamatis Kavadias, Manolis Katevenis, Dionisios Pnevmatikatos and Xiaojun Yang. FPGA Implementation of a Configurable Cache/Scratchpad Memory with Virtualized User-Level RDMA Capability. *Proc. IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS2009)*, July 2009.

[8] K. Gharachorloo, M. Sharma, S. Steely, and S. Van Doren. Architecture and design of alphaserver gs320. *SIGPLAN Not.*, 35(11):13–24, 2000.

[9] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 297–307, New York, NY, USA, 2008. ACM.

[10] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally. Architectural support for the stream execution model on general-purpose processors. *16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 3–12, 15-19 Sept. 2007.

[11] IBM. *PowerPC 750GX/FX Cache Programming*, Dec 2004.

[12] Intel. *Intel XScale Microarchitecture Programmers Reference Manual*, Feb 2001.

[13] M. Katevenis. Interprocessor communication seen as load-store instruction generalization. In K. B. e.a., editor, *The Future of Computing, essays in memory of Stamatis Vassiliadis*, pages 55–68, Delft, The Netherlands, Sept. 2007.

[14] S. W. Keckler, A. Chang, W. S. Lee, S. Chatterjee, and W. J. Dally. Concurrent event handling through multithreading. *IEEE Trans. Comput.*, 48(9):903–916, 1999.

[15] D. Koufaty and J. Torrellas. Comparing data forwarding and prefetching for communication-induced misses in shared-memory mps. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 53–60, New York, NY, USA, 1998. ACM.

[16] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[17] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):358–368, 2007.

[18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[19] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[20] J. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec. 1995.

[21] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[22] D. K. Poulsen and P.-C. Yew. Data prefetching and data forwarding in shared memory multiprocessors. *icpp*, 2:276–280, 1994.

[23] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D. I. August, and G. Z. N. Cai. Support for high-frequency streaming in cmps. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 259–272, Washington, DC, USA, 2006. IEEE Computer Society.

[24] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 214–224, New York, NY, USA, 2000. ACM.

[25] M. Reilly, L. C. Stewart, J. Leonard, and D. Gingold. A new generation of cluster interconnect.

[26] Saman P. Amarasinghe and Michael I. Gordon and Michal Karczmarek and Jasper Lin and David Maze and Rodric M. Rabbah and William Thies. Language and Compiler Design for Streaming Applications. *International Journal of Parallel Programming*, 33(2-3):261–278, 2005.

[27] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. *SIGPLAN Not.*, 29(11):297–306, 1994.

[28] S. L. Scott. Synchronization and communication in the t3e multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5):26–36, 1996.

[29] M. Wen, N. Wu, C. Zhang, Q. Yang, J. Ren, Y. He, W. Wu, J. Chai, M. Guan, and C. Xun. On-chip memory system optimization design for the ft64 scientific stream accelerator. *IEEE Micro*, 28(4):51–70, 2008.