Int J Parallel Prog
DOI 10.1007/s10766-011-0173-6

Cache-Integrated Network Interfaces: Flexible On-Chip Communication and Synchronization for Large-Scale CMPs

Stamatis Kavadias · Manolis Katevenis · Michail Zampetakis · Dimitrios S. Nikolopoulos

Received: 2 November 2010 / Accepted: 13 May 2011
© Springer Science+Business Media, LLC 2011

Abstract Per-core scratchpad memories (or local stores) 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, 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 network interface (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 technique that enables software configurable communication and synchronization primitives. We present three event response mechanisms that expose NI functionality to software, for multiword transfer initiation, completion notifications for software selected sets of arbitrary size transfers, and multi-party synchronization queues. We implemented these mechanisms in a four-core FPGA prototype, and measure the logic overhead over a cache-only design for basic NI functionality to be less than 20%. We also evaluate the on-chip communication performance on the prototype, as well as the

All the authors are member of HiPEAC.

S. Kavadias (✉) · M. Katevenis · M. Zampetakis · D. S. Nikolopoulos
Foundation for Research & Technology - Hellas, Institute of Computer Science (FORTH-ICS), Heraklion, Crete, Greece
e-mail: kavadias@ics.forth.gr

M. Katevenis
e-mail: kateveni@ics.forth.gr

M. Zampetakis
e-mail: mzampet@ics.forth.gr

D. S. Nikolopoulos
e-mail: dsn@ics.forth.gr

Published online: 01 June 2011

performance of synchronization functions with simulation of CMPs 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.

Keywords Cache · Network interface · Explicit communication · Synchronization

1 Introduction

As the number of processing cores per chip increases, so does the need for efficient and high-speed communication and synchronization support, to minimize overheads and enable applications that efficiently 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 increasing communication overheads.

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" in the literature) naturally lend themselves 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 scratchpad memories, enables the use of producer-initiated communication, as well as overlapping of communication with computation without the need for complex out-of-order processors and non-blocking caches. Software can exploit these opportunities in emerging runtime systems (e.g. Sequoia [6]), and when communication can be orchestrated appropriately for a given application.

In this paper, we present the architecture of cache integrated network interfaces that we developed in the context of the "SARC" project [30], to achieve the best of two worlds: the flexibility of caches and the optimization enabled by scratchpad memories. The proposed architecture allows sharing on-chip SRAM, at cache-line granularity, for caching, scratchpad, and network interface (NI) communication functions, all mapped in the application's virtual address space for virtualization. We support load/store access to remote scratchpads, and RDMA-copy operations that can be *explicitly acknowledged*. These communication mechanisms use virtual source and destination addresses and thus provide the equivalent of generalized read and write accesses for explicit communication.

In addition, we introduce event responses as a technique that enables efficient cache-integration of NI functionality. Event responses leverage line tag and state lookup in the normal cache access flow, for software-configurable communication and synchronization mechanisms. Three event response based primitives are discussed: *command buffers* used to initiate multi-word communication; *counters* that provide the equivalent of

memory barriers for explicitly-selected "accesses" of arbitrary size; and *queues* implemented in scratchpad memory for multi-party synchronization, supporting a single or multiple readers.

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 enable 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 also enable the maximal utilization of the available on-chip and off-chip memory bandwidth, with as little as 3 KB of buffering space 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 [25] that use coherently cached variables.

The rest of this paper is organized as follows: In Sect. 2 we discuss some background for this study and review related work. Section 3 presents our architecture in detail. In Sect. 4, we briefly describe our hardware prototype and report on the logic overhead of NI integration inside a cache. Section 5 presents performance evaluation on the prototype and with simulations, and Sect. 6 concludes the paper.

2 Background

This paper extends on our previous work in [17]. Here, we elaborate on the architecture of cache-integrated network interfaces and the technique of event responses that enables their efficient implementation, and also measure the logic overhead of NI integration inside a cache. In addition, the following subsection unfolds the insite behind our selection of previously proposed communication and synchronization primitives, and the design of novel ones.

2.1 RDMA, Queues, and Counters

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. Finally, 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 Sect. 3.4.

Single-reader queues, with similar basic functionality to hardware queue support in the Cray T3E [32] or mailboxes in the Cell BE [13], can be used to optimize many-to-one communication via multiplexing of senders. Forming multiple such queues in scratchpad memory also allows demultiplexing of message categories, similarly to register-interfaced queues in Tilera's TILE64 chip [35]. Compared to RDMA, single-reader queues have two disadvantages, making them more suitable for control information exchange than actual data transfers: (i) they require copying of data out of the queue and into the program's data structures,[1] and (ii) they fix data processing order to the unpredictable order of data arrival. Single-reader queues also have an advantage compared to RDMA: they require constant time for locating arrived input. RDMA requires per sender buffering at the receiver, and thus locating arrived input requires polling time proportional to the number of possible senders.

In addition to the above, although we do not implement or evaluate such an optimization, single-reader queues can support negative acknowledgments (NACKs) when there is insufficient space for arriving messages, to economize on receiver buffer space, irrespective of the number of possible senders. Such an optimization, enables receiver queue space that is proportional to the number of *anticipated* senders in a usual time interval, instead of space proportional to the *maximum* possible number of senders that is required for RDMA.

Single-reader queues are very closely-related to producer-consumer communication, in one-to-one, many-to-one, or many-to-many patterns. Philosophically, one can view counters and multiple-reader queues, that will be described in Sect. 3.3, as abstractions of different attributes of single-reader queues; multiple-reader queues abstract the reception order imposed by queue implementations, making them suitable for multiple concurrent readers, while counters abstract away the data buffered in a single-reader queue, and only accumulate "arrival events".

2.2 Related Work and Contributions

Ranganathan et al. [28] have proposed 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 of caches (further 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 (Sect. 3.2) in addition to locking lines in the cache.

Syncretic adaptive memory (SAM) [34] 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

[1] This disadvantage is less important in the case of queues in processor-integrated NIs, were data appear directly in processor registers.

to cache and main memory locations. Compared to our architecture, SAM requires a specialized compiler and can exploit cache-integration with a SRF using a specialized stream processor. In addition, SAM does not provide event response support.

In smart memories [7] 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 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.[2] It should be possible to support coherent cache and scratchpad organizations simultaneously and potentially program smart memories for event responses with adequate microcode memory, though the authors do not consider such optimizations. SiCortex [29] 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 have the most similarities with our cache-integrated NI, but our work focuses on keeping the NI simple enough to integrate with a high performance cache.

Prior work on fine-grain access control [31] 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 [32] supported single-reader queues in main memory and barrier/eureka FSM-based synchronization hardware. Our single-reader queues are optimized for an on-chip environment and enable multiple item granularities for use with multi-word short messages. Our counter-based barrier support is more general and easier to virtualize.

The MIT Multi-ALU Processor (MAP) [18] provided support for direct message transmission/reception from registers and dedicated receive side buffering. Our design avoids communication arrival interrupts required in MAP, exploiting virtualized destinations and explicit acknowledgments. The TILE64 chip [35] allows operand exchange via registers. A small set of queues is associated with registers, supporting settable tags that are matched against sender-supplied message tags, and a catch-all queue is provided for unmatched messages. All queues can be drained and refilled to and from off-chip memory. Our virtualized single-reader queues, formed in scratchpad memory, enable reception of direct transfers, sharing the fast and usual path through the processor's cache, without occupying processor registers. Cyclic buffering is enabled by updating a hardware visible head pointer, and multiple item granularities are supported.

Leverich et al. [21] 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

[2] 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.

reduction, which together with event responses and NI cache integration are the focus of our work.

Streaming hardware support, for general purpose systems using caches, was considered in [4,9,10,27]. 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 uniprocessor 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 [27], study hardware support alternatives for pipelined streaming. They propose a set of optimizations that can reach the performance of heavyweight hardware support. These optimizations include write-forwarding [1,19,20,26] at line boundaries, synchronization counters in L2 caches (which they do not describe), and small dedicated receive-side caches for pipelined streaming data in a separate address space. Our design integrates equivalent mechanisms inside general purpose caches, augmented with RDMA for efficient bulk transfers.

3 Cache-Integrated Network Interface Mechanisms

Explicit communication and synchronization mechanisms 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: *hardware access control* (e.g. Translation Lookaside Buffer—TLB), to mark virtual address regions as explicitly managed or scratchpad (Fig. 1), and *cache line state bits*, to indicate different access semantics and cache behavior (Fig. 2).

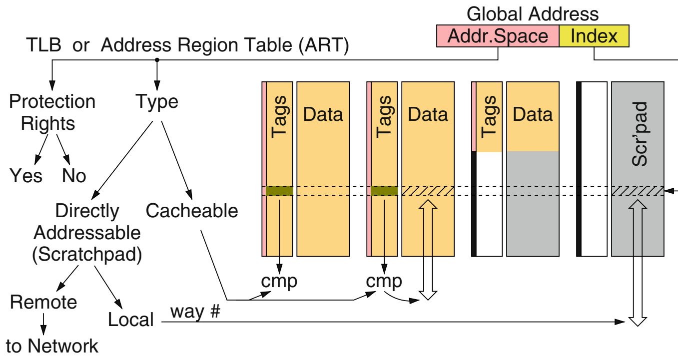

Fig. 1 *Memory access flow*: access control hardware provides type and "way" number information for directly addressable (*scratchpad*) regions

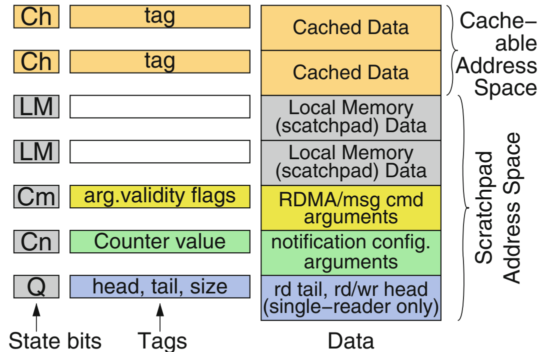

Fig. 2 *Cache line types*: state bits mark lines with scratchpad memory and communication semantics

As shown in Fig. 1, we assume that the access control mechanism[3] indicates in some way whether an address region or page contains cacheable or directly-addressed (scratchpad) data. Identifying local and remote scratchpad regions enables direct addressing, which locates scratchpad lines inside a cache using only a "way" number and the address index. Either we augment per region information in the access control mechanism with type and "way" number bits, or a region's physical address can include them. This obviates tag bit comparison to verify that a memory access actually hits into a scratchpad line; in this way, tag bits of scratchpad areas are freed, and we use them for other purposes in the case of communication semantics. Furthermore, indication of *remote scratchpad regions* identifies accesses that do not request local caching and the relevant space allocation; note that *remote stores*, potentially using *write-combining buffer(s)*, provide a very efficient method for transfering data and synchronization signals between producers and consumers.

In addition, to enable in-cache scratchpad allocation at cache-line granularity, circumventing the coarse granularity normally used for protection, we mark lines in

[3] In our SARC project, we use *Address Region Tables (ART)*, instead of the traditional TLBs, in order to support scalable page migration (only local ARTs are updated on local migrations –not all TLB's throughout the entire system), as explained in [16]; that issue, however, is orthogonal to what we discuss in this paper.

scratchpad regions, using line state bits in the cache, to support two functions: (i) locking of scratchpad lines in the cache, specifying them as *non-evictable* in the replacement mechanism; (ii) preventing intervention to the hit/miss calculation of cacheable space accesses, by ignoring tag matching for scratchpad lines. 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.

Moreover, the multiple *virtualized* communication *control/status "registers"* that we desire, are placed in scratchpad regions and utilize state and tag bits to support their semantics. As shown in Fig. 2, other than plain scratchpad memory, we mark in their state bits three types of special scratchpad lines, called *event sensitive lines (ESLs)*, to indicate their special semantics: *(i) command buffers* used for short message or *RDMA* command/status; *(ii) counters*, used for synchronization and notification through atomic increment operations; or *(iii) queue descriptors*, used 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 Sect. 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, multiple communication "registers" (ESLs) can be allocated in the (virtual) address space of any process, and 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 processor 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, the OS has to either know their address and type beforehand, or else it can discover them by reading the state and tag bits. To support the latter, our prototype maps these bits to a special address range.

Because local and remote ESL accesses may initiate communication, and to allow multiple overlapped such transfers, the NI must utilize a *job-list* to keep track of in-progress transfers and associations with ESLs. This job-list can be a simple hardware FIFO, that may support recycling of job descriptions, to avoid blocking short transfers behind potentially long RDMAs. The size of such a job-list can be scaled with small complexity increase, supporting the allocation of a large number of ESLs and allowing multiple outstanding transfers. Alternatively, the NI job-list can be implemented as a linked list of event sensitive lines, which should then include a next pointer field.

3.2 Event Responses

Event responses is a technique that exploits tagged memory to enable software configurable functions, extending the usual transparent cache operation flow, in which line state and tag lookup guides miss handling with coherence actions. Local or remote accesses to ESLs (NI events) are monitored, to atomically update associated NI

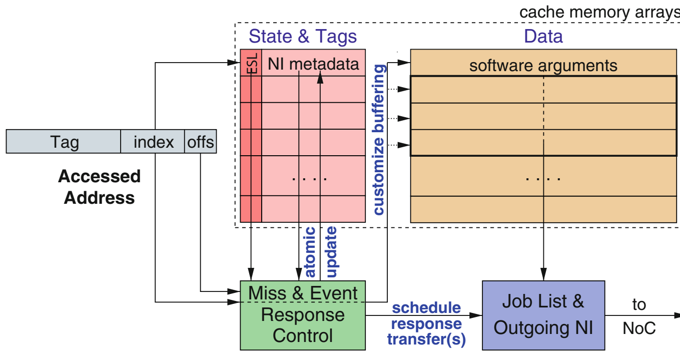

Fig. 3 Event response mechanism integration in the normal cache access flow. ESL accesses are the monitored events. In response, the NI can atomically update metadata in the ESL tag, modify the accessed index for custom buffering in scratchpad memory (e.g. managing some *lines* as a queue), or conditionally initiate response transfer(s). The outgoing NoC traffic controller can read transfer-related software configuration arguments from the ESL data block

metadata. Conditions can be evaluated on NI metadata, depending on the event type (e.g. read or write), to associate the effect of groups of accesses with communication initiation, or to manage access buffering in custom ways. These conditional NI actions are called event responses, and allow the implementation of different memory access semantics.

On every access to the cache (local or remote), normal cache operation checks the state and tag bits of the addressed line. As illustrated in Fig. 3, the NI monitors ESL access, atomically reads and updates associated metadata stored in the ESL tag, and checks whether relevant conditions are met. The state accumulated in the ESL tag can be used to customize the location of buffering, forming for example queues in scratchpad memory. When communication triggerring conditions are met, communication is scheduled by enqueueing a job description in the network interface job list. Software communication arguments can be pre-configured in the ESL data block. Such arguments are read and utilized by the outgoing traffic NI controller, when the relevant job description appears at the top of the job-list.

Event responses provide a framework for hardware communication and synchronization mechanisms configurable by software. The mechanisms designed, utilize appropriate ESL and scratchpad region internal organization, to support atomic operations in the NI that conditionally initiate communication. The different operations are designated by ESL state corresponding to the intended access semantics. They allow the description of multi-word communication arguments to the NI, enabling the NI communication functions, and the implementation of our synchronization primitives.

3.3 Communication and Synchronization Primitives

Four event-response mechanisms are designed, for command buffers, counters, single- and multiple-reader queues. Command buffers are used to send short messages, or initiate RDMA-copy operations. An opcode, size, destination and acknowledgement

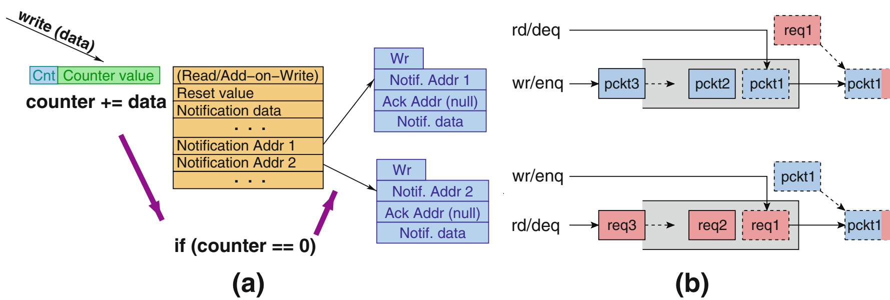

Fig. 4 Counter (**a**), and multiple-reader queue (**b**) operation

addresses are provided for messages, and data that may add up to the cache line size. In the case of RDMA-copy operations, a source address is provided instead of the data. The acknowledgement address is utilized for explicit acknowledgements, discussed in the next subsection. Command buffers exploit operation opcode and message size to *automatically* initiate a transfer when its description is complete (assuming all transfer arguments are written to a command buffer exactly *once*). On our prototype, RDMA-copy operation initiation requires only four store instructions.

Command completion is monitored at word granularity, using *validity flag* bits, kept in the (otherwise unused) tag of the command buffer, and allowing arbitrary order of the relevant stores, that can be separated by any time distance. The ability to tolerate arbitrary ordering of stores when describing a transfer, allows compiler reordering of instructions and store issue optimizations for weakly ordered memory accesses. The automatic detection of a complete transfer description obviates restrictions related to load and store reordering and the need for an additional access, required for explicit initiation. Multiple threads can be writing command buffers within their protection domain, concurrently, in user-mode.

Counters are intended to provide software notification(s) regarding the completion of an unordered sequence of operations (e.g. multiple transfer reception, or arrivals at a barrier). Counter function, supporting an atomic add-on-store operation, is depicted in Fig. 4a. The counter value is kept in the ESL tag, and can be accessed indirectly via offset zero of the ESL data block. An implementation dependent number of configurable notification addresses (four on our prototype) is provided in the counter ESL. When the counter becomes zero, a pre-configured word is sent to all notification addresses, by enqueueing corresponding job descriptions in the network interface job list. In addition, the counter is reset to a value also configured in advance, in the ESL data-block.

Single- and multiple-reader queues keep head and tail pointers in the ESL tag. The queue is formed in scratchpad memory adjacent to the ESL. Queue head and tail pointers are used to modify some of the least significant index bits, in order to access a queue entry in the data array, as shown in Fig. 3. Different offsets of the ESL data-block allow writing to the single-reader queue (sr-Q), reading the tail pointer, as well as reading and updating the head pointer, where the latter two operations are intended for the local processor. Data arriving to an sr-Q are written to the queue offset pointed

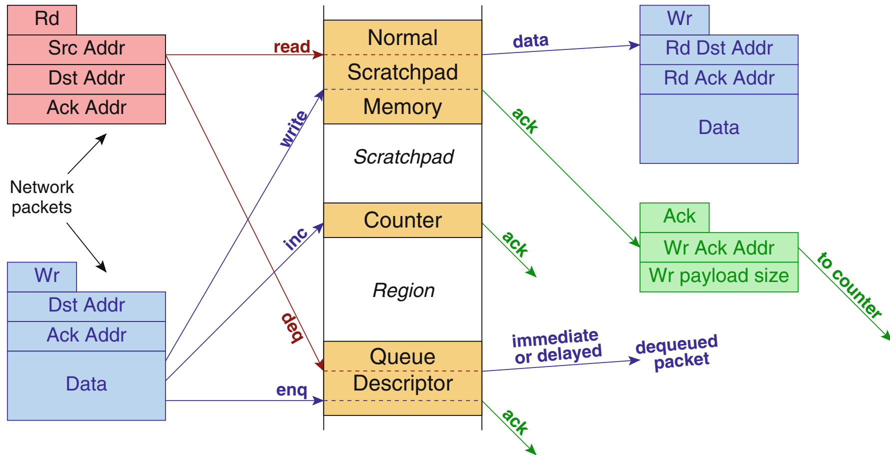

Fig. 5 Remote access to scratchpad regions and generation of explicit acknowledgements

by the queue tail pointer, atomically incrementing it. Items of power of two size (in words) are supported for queue access via short messages.

Multiple-reader queues combine multiplexing of short message data from multiple producers and request messages from multiple consumers, buffering either in the same queue, or, more accurately, in a pair of queues overlapped in the same scratchpad space. The mr-Q allows dequeue (read) operations that wait until data are enqueued (written), 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, enqueueing a job description in the NI job list, to sent the data of the write to the response address of the read.

Conceptually, the multiple-reader queue (mr-Q) buffers *either* data *or* requests. When data are buffered and a request arrives, shown in the upper part of Fig. 4b, the data on the top of the queue are matched with the request. When requests are buffered in the queue and data arrive, shown in the lower part of the figure, the data are matched with the top request in the queue. In either case, when a new item arrives (request or data), that is of the same type to those already buffered in the queue, it is also stored at the tail of the queue; this is also the case with arrivals at an empty queue.

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 5 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

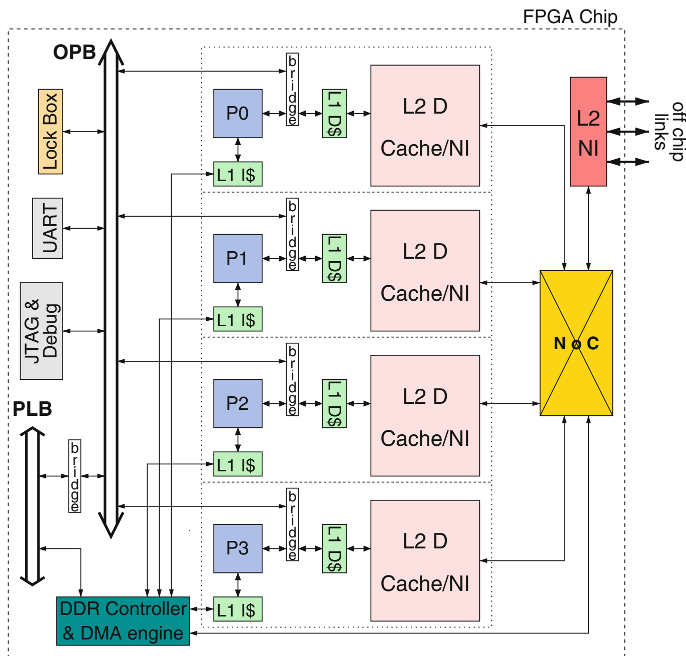

Fig. 6 FPGA prototype system block diagram

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, with the size of the written 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 The Hardware Prototype

We have fully implemented the architecture described in Sect. 3 in a hardware prototype based on a Xilinx Virtex-5 FPGA. A previous version of the prototype was presented in [14]. The current version is a major rewrite of the code, carefully pursuing logic reuse, implementing event responses, three levels of NoC priority and some other features not present in the version of [14]. The current version of the prototype was presented in [15], and we report here on the reduced logic cost, that resulted from some *alignment* optimizations in that version.

The block diagram of the FPGA system is presented in Fig. 6. There are four Xilinx microblaze IP cores, each with 4 KB L1 instruction and data caches and a 64 KB 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

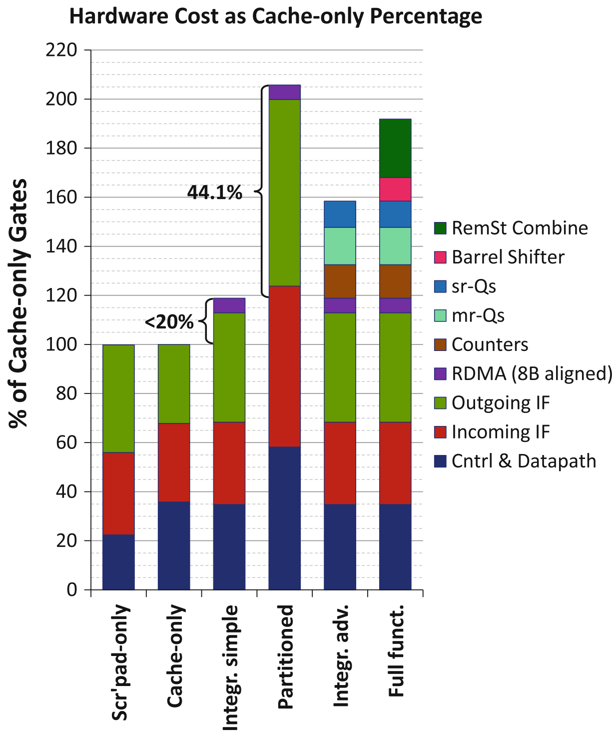

Fig. 7 Area optimization as a percentage of a cache-only design

DRAM controller –to which we added a DMA engine– and to the interface (L2 NI) of a future off-chip interconnect, over 3 high speed RocketIO transceivers, intended for a multi-chip system. All processors are directly connected over a bus (OPB) to a hardware *lock-box* supporting a limited number of locks, that we used for comparison purposes (Sect. 5.4.1).

4.1 Logic Cost of Cache-integrated NI Mechanisms

At a high level, event responses match the operation of NI communication and synchronization mechanisms to the controller and memory organization of a cache. In order to optimize our cache-integrated design at a lower level, we carefully adapt the format of transfer descriptions via command buffers to the NoC packet header format, so that we can exploit corresponding field alignment. This approach saves about 19.3% of the logic in our integrated design, over our measurement reported in [15]. In the results presented, we assume that 1 FPGA lookup table (LUT) or 1 flip-flop is equivalent to 8 gates.

Figure 7 compares the logic of our optimized cache-integrated design with a cache-only design and a partitioned one. The measurements are shown as a percentage of the base cache-only design (second bar from the left), and refer to controller and data-

path logic only, excluding the SRAM arrays for cache/scratchpad state, tags, or data. The simple integrated design (third bar from the left), which includes support for 8B-aligned RDMA, is less than 20% larger than the plain cache, and saves about 44.1% of the logic required for a partitioned cache-scratchpad design (fourth bar from the left). In addition, Fig. 7 shows the cost of adding counters ($\sim$13.6%), multiple-reader queues ($\sim$ 15.2%), and single-reader queues ($\sim$ 10.6%), for an advanced integrated design (second bar from the right). The relative cost of the full functionality implemented in our prototype is also shown (first bar from the right), which includes a barrel shifter for arbitrary RDMA alignment ($\sim$9.6%) and a remote store combining buffer ($\sim$23.8%).

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 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 *syslib*, *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. *Scrlib* supports an SHMEM-like programming model [33], where global addresses are given as a local-address/node ID pair. This allows us to avoid the need for a compiler that supports threads and shared addresses, that is not available for the FPGA environment. It also includes functions for marking a cache line as a command buffer, a counter, or a queue. Last, *nilib* contains functions for preparing and issuing DMAs, for managing command buffers, notifications, and queues, and for sending messages to remote scratchpad memories.

5.2 Simulation Infrastructure

In this paper, we model event responses in GEMS [23] and simulate counter-based barriers and mr-Q based locks for up to 128 processors. Our simulations are based on the GEMS memory system simulator, driven by accesses from the Simics [22] 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 Sect. 5.1 were 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.

5.3 Applications and Benchmarks

The STREAM triad benchmark [24] 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 the 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 [2] 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 [2]. 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 [25], 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.

5.4 Results

5.4.1 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 Fig. 6, whereas the second uses multiple reader queues (see Sect. 3). For barrier synchronization, we developed a barrier using the mutex lock implemented with the hardware lock box and a barrier using counters.

Table 1a 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 1b 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 8 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.

Table 1 Different lock and barrier latencies

	Mutex lock			MRQ lock		
CPUs	1	2	4	1	2	4
(a) Contended lock times						
Acquire	44	87	201	42	82	193
Release	32	32	32	21	21	21
Total	76	119	223	63	103	214
	Mutex barrier			Counter barrier		
CPUs	1	2	4	1	2	4
(b) Barrier times						
Cycles	188	281	618	75	105	117

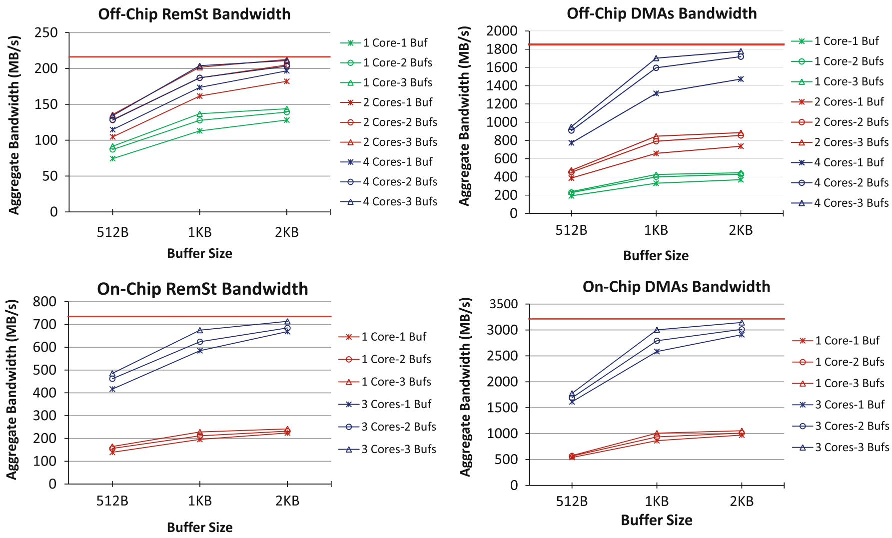

Fig. 8 Performance of STREAM triad benchmark

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 $\times$) 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 (3–4 KB) for buffering data in scratchpad memory.

Figure 9 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 perfor-

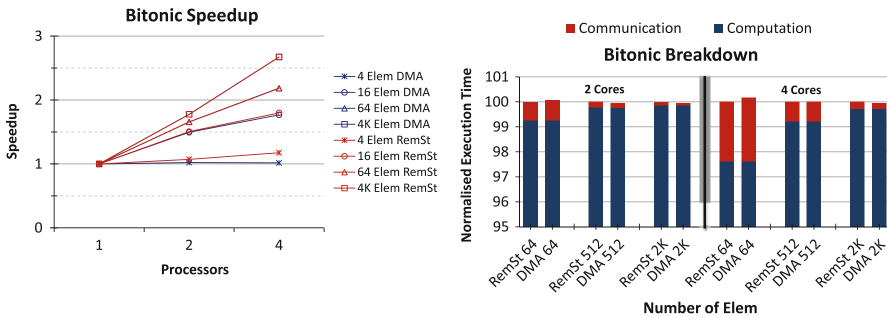

Fig. 9 Performance of bitonic sort

mance with DMAs exceeds only marginally performance with remote stores (by no more than 0.2%) due to the low communication to computation ratio 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 10 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.

5.4.2 Simulated Locks and Barriers

Figure 11a 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 contended lock-unlock operations for MCS locks will incur 3 to 5 misses per iteration, requiring remote acquisition of cache lines through the directory.

Figure 11b 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

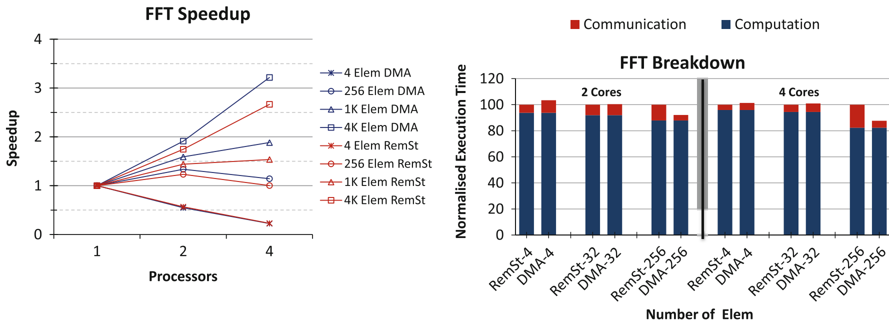

Fig. 10 Performance of FFT

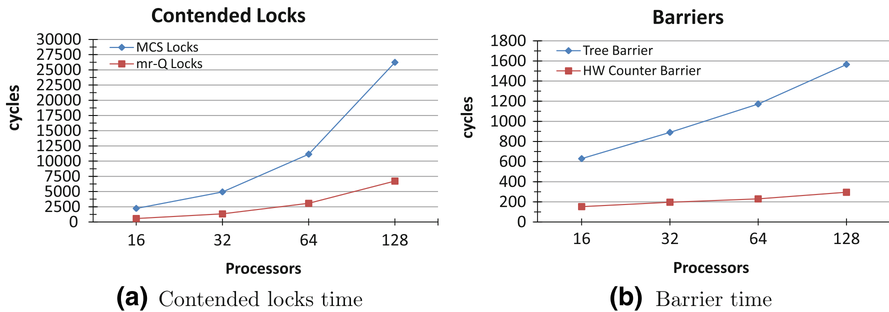

Fig. 11 Average latency versus number of processors for simulated barriers and locks. MCS locks and Tree-based Barriers use coherent communication. mr-Q Locks and Counter Barriers utilize event responses and direct scratchpad-to-scratchpad communication

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 length. For the benchmarks used, the communication efficiency of RDMA quickly dominates over remote stores for relatively small data transfers. Finally, RDMA transfers can saturate the on-chip bandwidth with as few as three overlapped transfers of 1 KB on our prototype. On our prototype, RDMA 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.

As CMP architectures become more distributed, mechanisms for on-chip direct communication may 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 for 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.

References

1. Abdel-Shafi, H., Hall, J., Adve, S.V., Adve, V.S.: 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, p. 204. USA, IEEE Computer Society, Washington, DC (1997)
2. Amarasinghe, S.P., Gordon, M.I., Karczmarek, M., Lin, J., Maze, D., Rabbah, R.M., Thies, W.: Language and compiler design for streaming applications. Int. J. Parallel Program. **33**(2–3), 261–278 (2005)
3. Bronevetsky, G., Gyllenhaal, J., de Supinski, B.R.: CLOMP: accurately characterizing OpenMP application overheads. In: Proceedings of the Fourth International Workshop on OpenMP (IWOMP), pp. 13–25. West Lafayette, IN (May 2008)
4. Cook, H., Asanović, K., Patterson, D.A.: 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. Falsafi, B., Lebeck, A.R., Reinhardt, S.K., Schoinas, I., Hill, M.D., Larus, J.R., Rogers, A., Wood, D.A.: Application-specific protocols for user-level shared memory. In: Supercomputing '94: Proceedings of the 1994 Conference on Supercomputing, pp. 380–389. IEEE Computer Society Press, Los Alamitos, CA, USA (1994)
6. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y. Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, p. 83. ACM, New York, NY, USA (2006)
7. Firoozshahian, A., Solomatnikov, A., Shacham, O., Asgar, Z., Richardson, S. Kozyrakis, C., Horowitz, M.: A memory system design framework: creating smart memories. In: ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture, pp. 406–417. ACM, New York, NY, USA (2009)
8. Gharachorloo, K., Sharma, M., Steely, S., Van Doren, S.: Architecture and design of AlphaServer GS320. SIGPLAN Not. **35**(11), 13–24 (2000)
9. Gummaraju, J., Coburn, J., Turner, Y., Rosenblum, M.: 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, pp. 297–307. ACM, New York, NY, USA (2008)

10. Gummaraju, J., Erez, M., Coburn, J., Rosenblum, M., Dally, W.J.: Architectural support for the stream execution model on general-purpose processors. 16th International Conference on Parallel Architecture and Compilation Techniques (PACT), pp. 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. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Dev. **49**(4/5), 589–604 (2005)
14. Kalokairinos, G., Papaefstathiou, V., Nikiforos, G., Kavadias, S., Katevenis, M., Pnevmatikatos, D., Yang, X.: FPGA implementation of a configurable cache/scratchpad memory with virtualized user-level RDMA capability. In: Proceedings IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS2009) (July 2009)
15. Kalokerinos, G., Papaefstathiou, V., Nikiforos, G., Kavadias, S., Katevenis, M., Pnevmatikatos, D., Yang, X.: Prototyping a configurable cache/scratchpad memory with virtualized user-level RDMA capability. Trans. HiPEAC (2010, to appear)
16. Katevenis, M.: Interprocessor communication seen as load-store instruction generalization. In: K. Bertels e.a. (ed.) The Future of Computing, Essays in Memory of Stamatis Vassiliadis, pp. 55–68. Delft, The Netherlands (Sept 2007)
17. Kavadias, S., Katevenis, M.G.H., Zampetakis, M., Nikolopoulos, D.S.: On-chip communication and synchronization with cache-integrated network interfaces. In: CF '10: Proceedings of the 7th ACM Conference on Computing Frontiers. ACM, New York, NY, USA (May 2010)
18. Keckler, S.W., Chang, A., Lee, W.S., Chatterjee, S., Dally, W.J.: Concurrent event handling through multithreading. IEEE Trans. Comput. **48**(9), 903–916 (1999)
19. Koufaty, D., Torrellas, J.: Comparing data forwarding and prefetching for communication-induced misses in shared-memory MPs. In: ICS '98: Proceedings of the 12th International Conference on Supercomputing, pp. 53–60. ACM, New York, NY, USA (1998)
20. Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W.-D., Gupta, A., Hennessy, J., Horowitz, M., Lam, M.S.: The stanford dash multiprocessor. Computer **25**(3), 63–79 (1992)
21. Leverich, J., Arakida, H., Solomatnikov, A., Firoozshahian, A., Horowitz, M., Kozyrakis, C.: Comparing memory systems for chip multiprocessors. SIGARCH Comput. Archit. News. **35**(2), 358–368 (2007)
22. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: a full system simulation platform. Computer **35**(2), 50–58 (2002)
23. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (gems) toolset. SIGARCH Comput. Archit. News. **33**(4), 92–99 (2005)
24. McCalpin, J.: Memory bandwidth and machine balance in current high performance computers. IEEE Comput. Soc. Tech. Comm. Comput. Archit. (TCCA) Newsl. (Dec 1995)
25. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. **9**(1), 21–65 (1991)
26. Poulsen, D.K., Yew, P.-C.: Data prefetching and data forwarding in shared memory multiprocessors. In: Proceedings of the 1994 International Conference on Parallel Processing (ICPP '94), vol. 2, pp. 276–280 (1994)
27. Rangan, R., Vachharajani, N., Stoler, A., Ottoni, G., August, D.I., Cai, G.Z.N.: Support for high-frequency streaming in CMPs. In: MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 259–272. IEEE Computer Society, Washington, DC, USA (2006)
28. Ranganathan, P., Adve, S., Jouppi, N.P.: Reconfigurable caches and their application to media processing. In: ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture, pp. 214–224. ACM, New York, NY, USA (2000)
29. Reilly, M., Stewart, L.C., Leonard, J., Gingold, D.: A new generation of cluster interconnect
30. SARC: Scalable computer ARChitecture: http://www.sarc-ip.org/. European IP Project (2005–2009)
31. Schoinas, I., Falsafi, B., Lebeck, A.R., Reinhardt, S.K., Larus, J.R., Wood, D.A.: Fine-grain access control for distributed shared memory. SIGPLAN Not. **29**(11), 297–306 (1994)
32. Scott, S.L.: Synchronization and communication in the T3E multiprocessor. SIGOPS Oper. Syst. Rev. **30**(5), 26–36 (1996)
33. Shan, H., Singh, J.P.: A comparison of MPI, SHMEM and cache-coherent shared address space programming models on a tightly-coupled multiprocessors. Int. J. Parallel Program. **29**(3), 283–318 (2001)

34. Wen, M., Wu, N., Zhang, C., Yang, Q., Ren, J., He, Y., Wu, W., Chai, J., Guan, M., Xun, C.: On-chip memory system optimization design for the FT64 scientific stream accelerator. IEEE Micro. **28**(4), 51–70 (2008)
35. Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.-C., III, J.F.B., Agarwal, A.: On-chip interconnection architecture of the tile processor. IEEE Micro. **27**(5):15–31 (2007)