Prototyping a Configurable Cache/Scratchpad Memory with Virtualized User-Level RDMA Capability

George Kalokerinos, Vassilis Papaefstathiou, George Nikiforos, Stamatis Kavadias, Manolis Katevenis, Dionisios Pnevmatikatos and Xiaojun Yang

Institute of Computer Science, FORTH,
Heraklion, Crete, Greece – member of HiPEAC
{george,papaef,nikiforg,kavadias,kateveni,pnevmati,yxj}@ics.forth.gr

Abstract. We present the hardware design and implementation of a local memory system for individual processors inside future chip multiprocessors (CMP). Our memory system supports both implicit communication via caches, and explicit communication via directly accessible local ("scratchpad") memories and remote DMA (RDMA). We provide run-time configurability of the SRAM blocks that lie near each processor, so that portions of them operate as 2nd level (local) cache, while the rest operate as scratchpad. We also strive to merge the communication subsystems required by the cache and scratchpad into one integrated Network Interface (NI) and Cache Controller (CC), in order to economize on circuits. The processor interacts with the NI at user-level through virtualized command areas in scratchpad; the NI uses a similar access mechanism to provide efficient support for two hardware synchronization primitives: counters, and queues. We describe the NI design, the hardware cost, and the latencies of our FPGA-based prototype implementation that integrates four MicroBlaze processors, each with 64 KBytes of local SRAM, a crossbar NoC, and a DRAM controller. One-way, end-to-end, user-level communication completes within about 20 clock cycles for short transfer sizes.

1 Introduction

Memory hierarchies of modern multicore computing systems are based on one of the two dominant schemes – multi-level caches, or directly-addressable local "scratchpad" memories. Caches transparently decide on the placement of data, and use *coherence* to support communication, which is especially helpful in the case of *implicit* communication, i.e. the input data-set and/or the producer of data are not known in advance. However, caches lack determinism and make it hard for the software to explicitly control and optimize data transfers and locality in the cases when it can intelligently do so. Furthermore, coherent caches scale poorly to over hundreds of processors. Scratchpads are popular in embedded [1] and special purpose systems [2][3], because they offer predictable performance – suitable for real-time applications – and also offer scalable general-purpose

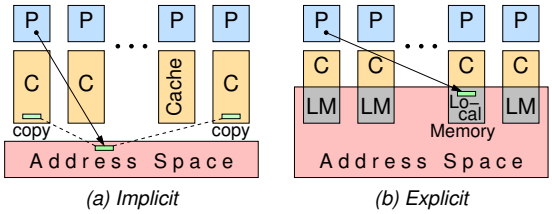


Fig. 1: Implicit vs. Explicit Communication

performance by allowing explicit control and optimization of data placement and transfers. *Explicit* communication uses *remote direct memory accesses (RDMA)*; it is efficient, and it becomes possible in the cases when the producer knows who the consumers will be, or when the consumer knows its input data-set ahead of time. Recent advances in parallel programming and runtime systems [4][5] allow the use of explicit communication with minimal burden to the programmers, who merely have to identify the input and output data sets of their tasks.

Our goal is to provide *unified* hardware support for *both implicit and explicit* communication within the same address space as shown in Figure 1. To achieve low latency, we integrate our mechanisms close to the processor - in the upper cache levels, unlike traditional RDMA that is implemented at the I/O bus level. We provide *configurability* of the local SRAM blocks that lie next to each core, so that they operate either as cache or scratchpad, or as a dynamic mix of the two. Configurability is at *run-time* allowing different programs with different memory requirements to run on the same core, or even different stages of a program to adapt the underlying memory to their needs. We also strive to merge the hardware required by the cache and scratchpad into one *integrated* Network Interface (NI) and Cache Controller (CC), in order to economize on circuits.

We propose a simple, yet efficient, solution for cache/scratchpad configuration at run-time and a common NI that serves both cache and scratchpad communication requirements. The NI receives DMA commands and delivers completion notification in designated portions of the scratchpad memory. This allows the OS and runtime systems to allocate as many NI command buffers as desired per protection domain, thus effectively virtualizing the NI, while providing user-level access to its functions so as to drastically reduce latency. We improve SRAM utilization compared to traditional NIs (that used dedicated memories) by sharing the SRAM blocks between the processor and the NI, and we sustain high-throughput operation by organizing these SRAM blocks as a wide interleaved memory. The scratchpad space can be allocated inside the L1 or L2 caches and consequently the NI is brought very close to the processor, reducing latency. Our NI also offers fast messages, queues, and counters, as *synchronization* primitives, to support advanced interprocessor communication mechanisms. We assume *Global Virtual Addresses* and *Progressive Address Translation* [6].

This article presents, in Section 2, the architecture of the proposed integrated memory hierarchy and NI, along with the supported synchronization primitives.

Section 3 describes the hardware implementation, through FPGA prototyping, of the configurable level-2 cache/scratchpad and the integrated CC/NI controller. We report on the hardware cost, showing that the merged NI and Cache Controller uses 35 percent less hardware than the two separate systems, in addition to the economy resulting from the shared SRAM space. We measure one-way, end-to-end, user-level communication to be about 20 clock cycles for short transfer sizes. We also evaluate the use of our primitives with software and present the performance benefits in a set of case studies, Section 4. Related work, conclusions, and future work appear at the end of the article.

2 Architecture Overview

Our proposed architecture targets chip multiprocessor systems with tens or hundreds of processor cores: each core has at least two levels of private caches and communicates with shared memory using Global Virtual Addresses [6]. This section describes run-time configuration of the local SRAM blocks as cache and/or scratchpad. We explain how scratchpad memory can be used to support virtualized NI command buffers, and present our hardware synchronization primitives.

2.1 Run-time configurable Scratchpad

Scratchpad space in our scheme is declared as a contiguous address range and corresponds to some cache lines that are pinned (locked) in a specific way of the cache, i.e. cache line replacement is not allowed to evict (replace) them.

Scratchpad areas can be allocated inside either L1 or the L2 caches. Most applications seem to require relatively large scratchpad sizes, so the L2 array is a more natural choice. Moreover, L2 caches offer higher degree of associativity, hence more interleaved banks. Although L2 latency is higher than L1, the performance loss due to this increased latency is partly compensated in two ways: *(i)* The L2 and scratchpad supports pipelined random accesses (read or writes) at a rate of 1 per clock cycle; *(ii)* configurable parts of the scratchpad space can be cacheable in the (write-through) L1 caches [1].

Owing to the use of progressive address translation [6], caches and scratchpad operate with virtual addresses, and the TLB only needs to be consulted when messages are transferred through the NI and the NoC to another node. In lieu of the processor-TLB, our architecture has a small table called Address Region Table (ART). As shown in Figure 2, ART provides a few bits that determine whether an address region contains cacheable or directly-addressed (scratchpad) data. This is important when remote scratchpad regions are addressed, so that the hardware accesses them remotely, rather than locally caching them. It also obviates tag bit comparison to verify that a memory access actually hits into a

[1] Write-back policy can also be used, provided that coherence between L1 and L2 is maintained. However, the write-through policy simplifies coherence without any performance loss. The inclusion property assumed here, is more intuitive than exclusion that would require moving locked lines between the cache levels.

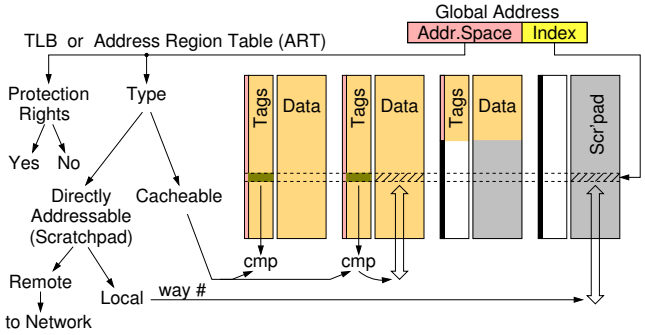


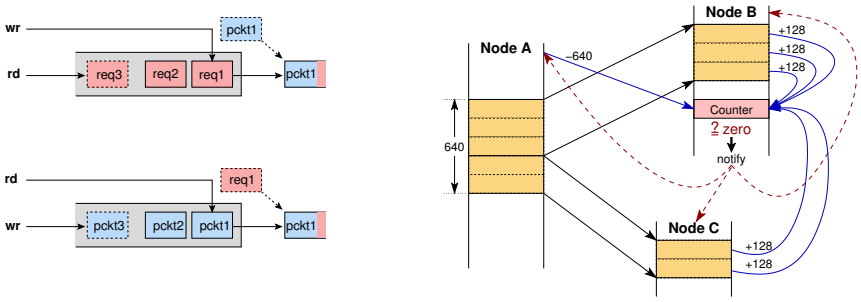
Fig. 2: Memory Access Flow

scratchpad line; hence, tag bits of scratchpad areas are freed, and can be used for other purposes, such as implementing communication semantics for RDMA commands, counters, and queues that will be described shortly. Regions marked as local scratchpad in the ART occupy a set of blocks in the data portion of an L2 memory way block, such that low-order bits (the cache index) are compatible with the scratchpad address. The region can be freely allocated into any of the cache "ways", with ART identifying the "way" used. Each of the blocks in the region is marked as non-evictable in its state bits. This marking allows the distinction of memory access semantics at cache block granularity, and is used to ignore the actual tag-matching of the hit logic, as well as to prevent replacements. This mechanism allows for run-time 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.

2.2 Virtualized user-level DMA

NI command buffers are DMA control areas which are allocated upon user software demand and reside in normal scratchpad regions. These buffers share the same ART entry with normal scratchpad and the distinction is made using a special bit (cache-line state), set upon allocation. Any user program can have dedicated NI command buffers (DMA registers) in its scratchpad region; this allows a low-cost virtualized DMA engine where every process/thread can have its own resources. To ensure protection of the virtualized resources, we also utilize permission bits in the ART and demand the OS/run-time system to update the ART appropriately on context switches. Moreover, the inherent support for dynamic number of DMAs at run-time promotes scalability and allows the processes to adapt their resources on the program's communication patterns that might differ among different stages of a program.

DMAs are issued as a series of store instructions – to provide the arguments: opcode, size, source and destination address – destined to words within command buffers, that gradually fill DMA command descriptors, possibly out-of-order. The NI uses a command protocol to detect command completion and



(a) Multiple-reader Queues (b) Multi-packet RDMA completion

Fig. 3: Illustrating advanced interprocessor communication primitives.

inform the DMA engine that a new command is present. All new and pending commands are kept in a *Network Job List* that is served by the NI according to its scheduling policy. When serving DMAs, the NI generates packets along with their customized lightweight headers. Packets belong to one of the two primitive categories: *Write* or *Read*. The NI carefully segments the DMAs into smaller packets when they exceed the maximum network packet size. The cache controller uses the *Network Job List* to request write-backs upon replacements and fills upon misses: the same mechanisms, and the same Read and Write packets, serve DMA transfers as well as cache operations.

2.3 Interprocessor Communication (IPC) Primitives

We provide advanced NI features that offer additional flexibility to the programmer in order to achieve more efficient communication between processors. We implement *Remote Stores* with write combining, to scratchpad regions of remote processors, in order to optimize remote access latency [7]; the ART can identify scratchpad ranges as remote. NI command buffers, described above, can also be used for fast *Messages*, allowing atomic, multi-word transfers. Message data are provided directly by the processor and no source address is needed. In addition, an explicit acknowledgment address can be specified to support software nofitication of transfer completion; acknowledgment addresses are allowed to be "null" to deactivate the mechanism. Multi-segment RDMA completion notification requires additional hardware support as described below.

We also provide *Remote Queues* as an appropriate level of abstraction for multiprocessor synchronization [8]. Queues are hosted inside scratchpad regions and their configuration (size, pointers and item granularity) can be programmed in the tags of special control lines. *Single Reader Queues* are provided to support efficient many-to-one control information exchange, with receiver polling to a single location. More advanced, *Multiple Reader Queues* (mr-Qs) are provided as a means for many-to-many synchronization, by allowing asynchronous write (enqueue) and read (dequeue) operations from any number of processors. As shown in Figure 3a, read requests arriving at an empty mr-Q are recorded,

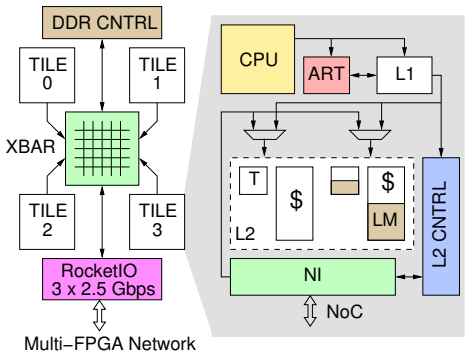


Fig. 4: FPGA Prototype Block Diagram

waiting until corresponding writes arrive, thus effectively *matching* read and write requests in time. Upon successful matching, a response packet is generated; matching is dual, i.e. either writes or reads might wait to be matched. Multiple reader queues can also be used for locks or to accelerate task/job dispatching.

Finally, we implement *Counters* with atomic add-on-store capability, also hosted in scratchpad space, as a primitive to support completion notification for an unordered sequence of operations, such as multiple RDMA transfer completion, barriers, and other synchronization operations. Counters are initialized with a value (e.g total transfer size in bytes) via a remote store and trigger single-word writes to notification addresses when they expire (reach zero). For RDMA transfer completion, software can specify an explicit acknowledgment address targeting a counter, which will gather all partial acknowledgments for DMA segments, as illustrated in Figure 3b. In the scenario shown, a single RDMA transfers 640 bytes. The destination region is mapped in the scratchpad of two separate nodes (nodes B and C). When all acknowledgments arrive at the counter, as well as the initialization value of -640, the counter triggers three notifications towards preconfigured addresses on nodes A, B and C. Counters is the only support required by the network interface for adaptive/multipath routing NoC optimizations, since RDMA transfer completion notifications will also work correctly with out-of-order packet arrivals. The only requirement for correct operation of the counter is that the NoC never generates duplicate packets.

3 FPGA-based Prototype and Implementation

Our hardware prototype is implemented in a Xilinx Virtex-5 FPGA using four MicroBlaze soft-cores as processors. The processors are 32-bit, in-order, and have a traditional 5-stage pipeline that also supports single-precision floating point operations. Each processor tile has a private data cache hierarchy, with a configurable L2 cache/scratchpad memory tightly-coupled with our NI. Instructions are fetched from private L1 instruction caches. The prototype is equipped with a 256MB DDR2 SDRAM which is used as main memory and is shared between

tiles. Communication between tiles and the on-chip DRAM memory controller is achieved through a 64-bit, 5-port crossbar switch (XBAR) that features three priorities and applies round-robin scheduling; contention-less crossbar traversal costs 1 clock cycle. An additional switch port can be used to provide multi-FPGA connectivity through multiple external high-speed serial links (RocketIO), and thus our modular design can be expanded with multiple boards in order to build larger scale systems. Cache coherence is not currently supported. The operating clock frequency of the system is currently 75MHz and its block diagram along with the major components is illustrated in Figure 4.

3.1 Configurable Cache/Scratchpad Memory

Every tile of our prototype implements a private data L1 cache and a private, configurable, data L2 cache/scratchpad. These are smaller than one would expect in a CMP, due to limited FPGA resources. Typically, L1 caches range from 16 to 64 KBytes, 2 to 4 way set associative, with 64-byte lines. Our implementation has scaled down the L1 caches to 4KB, direct-mapped, with 32-byte cache-lines. L1 caches are write-through, with 256-bit wide (one cache line) refills, a single-cycle hit latency, and follow "no-allocate" policy on store misses. L2 caches, on the other hand, are usually much larger, with sizes beyond 1MB, associativity up to 16-ways, and line size of 64 bytes or more. Scaling down again, we have designed a 64 KB, 4-way set-associative write-back L2 cache with 32-byte lines. Our L2 cache supports multiple hits under a single miss in order to minimize processor idle time. The L2 controller serves write-backs and fills on misses, using the transfer primitives of the tightly-coupled NI as described below.

The key component that allows us to configure and use parts of the L2 cache as scratchpad is the Address Region Table (ART); its function is similar to a traditional TLB, but it provides only protection and type information – not physical address translation– hence the ART can be smaller than a TLB (and have no misses), because it can describe potentially huge regions of the address space in each entry. The ART classifies each memory access as one of: *(i)* cacheable, *(ii)* local scratchpad, *(iii)* remote scratchpad, *(iv)* tag access (used to access and set lock bits in L2), or *(v)* register access (NI control registers that customize specific features). The ART is placed in parallel with the L1 cache, and is probed on every memory access of the processor; a copy of the ART is also used by the incoming NI. Routing in our prototype is based on physical addresses, thus we use a static mapping: each L2 data and tag array has a unique physical address (nodeID and way number are encoded in the address bits).

An important issue for the efficient use of scratchpads and their associated DMAs is the available memory bandwidth. Scratchpad areas in our design are hosted inside the L2 memory banks and the NI accesses them at high rate when performing DMAs. On the other hand, the default set-associative cache organization would require all the ways (tags + data) to be probed in parallel, causing conflicts and thus limiting the available data array bandwidth for the NI. In order to reduce the bandwidth required by the typical L2 cache operation and use it more efficiently for NI operations, we implement a phased L2 cache:

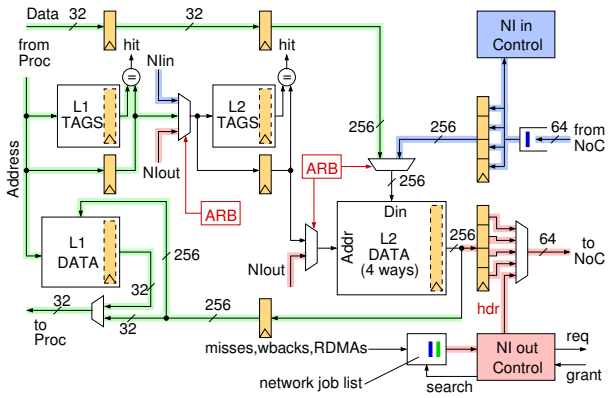


Fig. 5: Cache/Scratchpad Pipeline

the tag arrays are accessed first and the data arrays are accessed only on hits. Our cache-line wide (256-bit) L2 data array, allows L1 misses to be served in a single clock cycle, thus we avoid occupying the data arrays for multiple cycles. The outgoing and incoming NI paths also access data in 256-bit chunks and since the NoC is 64-bit wide, the maximum access rate per path is 1 per 4 clock cycles. As a result, more than 50% of L2 cycles is guaranteed for L1 requests.

Figure 5 presents the datapath and the pipeline of our design. All memory accesses arriving from the processor are checked against the ART regions and probe the L1 cache. Hits are served normally, while misses, stores, scratchpad and tag accesses, are sent to L2 along with all required control information: type of access and way (if scratchpad). In the first cycle, the L2 controller arbitrates among requests from NI in, NI out and the L1, and probes the tags. In the next cycle, the selected agent accesses the data of a specific cache way. To reduce scratchpad access latency, scratchpad lines are L1-cacheable. The L2 controller keeps the cached scratchpad lines coherent by issuing local invalidations when writes arrive from remote nodes; no further coherence actions are required since the L1 is write-through. Scratchpad loads that miss in the L1, have a minimum latency of 4 clock cycles, while stores take 3 clock cycles to reach the L2. The observed processor latency for stores is 1 clock cycle, stores are immediately acknowledged and propagate in the pipelined memory hierarchy.

3.2 NI Operation and Mechanisms

The NI is tightly-coupled to the L2 cache and serves all transfers from/to the tile's memory and the NoC. The heart of the outgoing NI path is the *Network Job List* which keeps the posted jobs that need to be served. The incoming NI path serves inbound traffic, stores data in-place and, depending on the type of traffic (cache or DMA), collaborates with the L2 controller to complete operations.

NI Command and Control Lines are allocated on software demand inside scratchpad areas, the state bits of locked lines distinguish them to four types:

- *Normal Memory:* normal scratchpad memory without side-effects.
- *Command Buffer:* are analogous to (virtualized) I/O command registers, and buffer RDMA and message requests. They are monitored by command completion hardware, which posts new jobs to the *Network Job List*.
- *Queue:* such cache lines contain metadata (pointers, size, item granularity) in the free tag part, describing a queue implemented as a circular buffer. The actual queue space is allocated separately, by software, inside scratchpad areas, outside the cache line itself. Two types of queues are supported: (i) *Single Reader Queues* (many-to-one) and (ii) *Multiple Reader Queues* (many-to-many). Single Reader Queues require one head and one tail pointer and the element size can be configured to 4-bytes, 8-bytes, 16-bytes, or a full scratchpad line. The head and tail pointers can be read via loads to specific block offsets. The Multiple Reader Queues have a fixed element size of 32-bytes and require one head pointer and two tail pointers: (i) a write-tail pointer for write packets (enqueues) and (ii) a read-tail pointer for read packets (dequeues). Incoming *write* packets (e.g. from remote store, message, or RDMA) destined to queue-type lines, are *enqueued* inside the circular buffer, and the NI controller updates the tail (or write-tail) pointer. Incoming *read* packets destined to Multiple Reader Queues record their response address in the queue body and update the read-tail pointer. Bound checking and pointer wrap-around is handled for head and tail pointers, as well as testing for queue full conditions. Matching a dequeue packet with an earlier enqueue packet (or vice-versa) is achieved by comparing the tail pointers with the head pointer and result in posting a new job in the *Network Job List*. The head pointer of Single Reader Queues is updated under software control while the head pointer of Multiple Reader Queues is updated by the NI when it completes the transfer associated with a *match* operation.
- *Counter:* these lines contain a 24-bit counter in the free tag part, and up to four notification addresses in the data part. Writes to word-offset zero increment the counter by the (signed) contents of the write. Upon reaching zero, the counter triggers the transmission of notification packets to the notification addresses by posting several jobs in the *Network Job List*.

Additionally, the NI serves incoming RDMA-Read requests. In order to meet the buffering requirements for incoming requests, without dedicating a separate memory block, we require the software to allocate a *Read Service Queue*, in the form of a Multiple Reader queue, and then assign its address to a special register.

NI Commands and Protocol: Commands to the NI are issued as a series of stores to the data part of Command Buffer lines. Our protocol defines two types of commands: *(i)* Copy and *(ii)* Message. Copy descriptors are DMAs and have a fixed size of four 32-bit words, while messages have any size up to one cache-line (eight 32-bit words in our prototype). In order to achieve automatic command completion, every descriptor should contain its own size (in bytes) inside the word at offset zero. The first word of every descriptor contains the following fields: *(i)* 8-bits descriptor size (bytes), *(ii)* 8-bit opcode (copy/message), *(iii)*

16-bit copy size (bytes - max 64 KBytes), used only when opcode is copy. For Copy descriptors this first word is followed by three mandatory virtual address arguments: *(a)* source, *(b)* destination, and *(c)* acknowledgment. For Message descriptors the first word is followed by two mandatory virtual address arguments –*(a)* destination and *(b)* acknowledgment– and up to five optional words that constitute the actual payload of the message. The NI uses its copy of the ART to distinguish local source addresses (write-RDMA) from remote sources addresses (read-RDMA), to validate (for protection purposes) the address arguments.

Completion Monitor: The NI includes a monitor circuit for command buffers, and uses the descriptor size to detect completion of commands, even in the presence of out-of-order stores, but assuming single-write of each word inside the command buffer line. The monitor is activated when stores arrive to cache-lines marked as command buffers, and a bitmap of the already completed words is formed and updated. The bitmap is kept in the free tag bits of these lines and when the number of consecutive "ones" matches those implied by the descriptor size, then command completion is triggered. Upon completion, a new job description containing the address of the command buffer is posted in the *Network Job List*. Since the completion bitmap is kept in the tags of each command buffer line, interleaved command issuing is supported offering full virtualization (e.g. threads can preempted while composing a command).

Remote Stores: Store instructions to addresses belonging to *remote scratchpad* regions (as identified by the ART), result in network packets carrying write requests, identical to RDMA or message packets (of data size 1 or more words). Stores marked as "remote" are kept in the *Remote Store Buffer*, and served by the outgoing NI engine as soon as it is free. A write-combining mechanism is implemented: if multiple remote stores to adjacent addresses arrive before some previous ones have departed, they are all coalesced in a single, multi-word-write packet. In order to support remote stores' completion notification, i.e. keep track if all remote stores have been successfully delivered, we use a special NI register that counts the total volume (in bytes) of departed remote store traffic; the acknowledgment address of remote store generated packets is automatically set to point to this register. Every time remote stores arrive in their destination(s), acknowledgment packet(s) that contain the delivered size are sent back to the sender in order to update the NI counter. The software can check the latter counter for *zero* to ensure that all remote stores have been successfully delivered.

Completion Notifications: We assume multi-path (adaptive) network routing, hence the multiple packets of a large RDMA may arrive out-of-order; the packet data will be written in-place, given that each of them carries its own destination address, but RDMA completion detection must now be performed by counting the number of bytes that have arrived (our network never generates duplicates). We implement counters to support RDMA completion notification. Each *session*, of one or more RDMA operations, uses one counter (allocated by software) as the acknowledgment address for its operations. The issuer decrements that counter by the total size of all RDMA transfers. Every RDMA packet

carries the counter address in its acknowledgment field; upon successful write, an acknowledgment is sent to the counter and increments it by the packet size. When the counter reaches zero the NI automatically sends notification packets to its pre-configured notification addresses.

Cache Transfer Support: The L2 cache controller issues requests for fills and write-backs by posting new job descriptions in the *Network Job List*. The job descriptions contain the appropriate opcodes and address: source address for a fill and destination address for a write-back. The outgoing NI uses the provided opcodes to format and generate the appropriate outgoing packets. The cache controller uses a Miss Status Handling Register (MSHR) structure, to keep track of outstanding write-backs and misses (transient cache-line states), and updates it appropriately when the requests are served by the NI. The number of supported outstanding cache misses is limited by the number of MSHRs; we currently support one outstanding miss.

Outgoing NI: The outgoing NI engine features a *Network Job List* in order to accept and manage requests for outgoing network operations. The sources of requests are typically the following:

- *L2 Cache Controller:* requests for write-backs and fills.
- *Completion Monitor:* explicit transfers, i.e. RDMAs and messages, when command completion is triggered for command buffers.
- *Counters:* up-to four completion notifications when a counter expires.
- *Multiple Reader Queues:* responses when enqueues and dequeues are matched.
- *Remote Store Buffer:* remote stores waiting in the remote store buffer.
- *Incoming NI engine:* remote acknowledgments from incoming packets.

Requests are posted in the *Network Job List* in the form of job descriptions. Each job description contains: (i) an opcode field that specifies how the arguments are interpreted and how the transfer should be handled by the outgoing NI engine, (ii) an address field that specifies either a local or a remote address (it may be a cacheable address, a command buffer, an acknowledgment, or a Multiple Reader Queue), (iii) the destination node number for the generated packet(s), (iv) the network priority (three available) of the packet(s) in order to avoid deadlocks of higher level protocols, e.g. cache coherence.

Upon receiving a job description, the outgoing NI first uses the destination node number to arbitrate for the NoC (request-grant protocol). When a network slot is granted the NI proceeds to the transfer, otherwise the current descriptor is recycled and put in the back of the *Network Job List*. The latter recycling tries to avoid head-of-line (HOL) blocking, when network destinations are congested, without requiring "expensive" per-output queues (VoQs). Recycling allows us to remove the outgoing per-priority network FIFOs, since pending transfers can wait inside the *Network Job List* and the packets need not be generated.

When a network slot is granted by the NoC, the NI operates in "cut-through" mode and generates packets – along with their customized lightweight headers and CRC checksums – that belong to one of the two primitive categories: *Write*

or *Read*. Cache write-backs, RDMA writes, messages, remote stores and acknowledgments belong to the *Write* category (carry data payload and acknowledgment address), while cache fills, RDMA reads and remote loads belong to *Read* category (carry the request arguments). Orthogonally to the primitive category, the packets in our prototype are sent with different network priorities as follows:

- *Low priority:* cache fills, RDMA reads, remote loads
- *Medium priority:* write-backs, RDMA writes, messages, remote stores.
- *High priority:* acknowledgments.

The payload of *Write* packets is acquired from the L2 data arrays, by iteratively reading chunks of 256-bits; the chunks are in turn serialized through the 64-bit NoC in four successive clock cycles. The NI segments large transfers, i.e. RDMA-Writes, into smaller packets when they exceed the maximum packet size (256-bytes in our prototype), or when alignment reasons dictate it. An RDMA transfer is served until it occupies a maximum network packet and then the corresponding job is recycled in the *Network Job List*; the associated command descriptor is also updated. Forcing large RDMA transfers to pause, offers fairness and reduces the latency of small packets that may wait behind large RDMAs. Moreover, the segmentation mechanism uses both source and destination addresses in order to generate packets that do not cross 256-byte boundaries. Additionally, our outgoing NI engine supports arbitrary source and destination address alignments (byte offsets) and leverages a barrel shifter to properly align and pad packets; the latter operation is only performed at the source nodes and thus the packets arrive to destinations nodes already aligned. When all packets of an RDMA transfer have been sent, the NI updates the actual command descriptor to signal local RDMA departure and allow the associated command buffer to be reused by software.

Incoming NI: The incoming NI exploits the header CRC contained in all packets and operates in "cut-through" mode to reduce latency. As soon as the header CRC is verified, i.e. destination address and packet size are correct, packets' payload can be safely delivered in memory without having to wait for body CRC verification; body CRC is carried in the last word of the packet. Upon reception, the NI writes the packets in per-priority network queues and notifies the incoming engine; network priorities are strictly served in descending order. The incoming NI engine gathers up-to four 64-bits words from the incoming network queues in order to create 256-bit chunks and write them at once in the wide L2 memory. The engine has first to identify whether a packet belongs to cache or scratchpad traffic, by checking the state bits of the destination address. If the destination is a cache-line waiting to be filled, then the NI delivers data in place and signals the L2 controller; only write-type packets are supported for incoming cache traffic. Write-type packets destined to lines in scratchpad space have to perform different steps according to the type of the line. In plain scratchpad lines, data are delivered in-place and an extra write with the packet size is performed to the acknowledgment address, if non-NULL. All writes from the incoming network, generate local invalidations to the L1 cache to ensure that no

Table 1: Hardware Cost Breakdown in FPGA Resources

Block	**LUTs**	**Flip Flops**	**BRAMs**
MicroBlaze + Instr. Cache	2712	2338	4
L1 + ART.	913	552	3
L2 Cntrl. + Arrays + Arb.	1157	893	23
NI Total	5364	2241	2
- Rem-Store Buff.	398	312	0
- Compl. Monitor	223	62	0
- Counters	286	99	0
- Queues	1011	45	0
- Outgoing NI	2042	1015	1
- Incoming NI	1404	708	1
Tile Total	10146	6024	34
NoC (5x5)	2820	750	0
DDR2 SDRAM Cntrl.	3745	4463	0
Total (4x Tile)	47149	29309	136

stale scratchpad data remain there. Incoming write packets destined to *Counter* lines are handled in an analogous manner; only their first word is considered. If a packet is destined to a *Queue*, then the queue descriptor is accessed and the appropriate tail pointer (read-tail for read packets and write-tail for write packets) is used to enqueue the incoming packet. Read-type packets carrying a DMA request use the queuing steps, mentioned before, to enqueue in the *Read Service Queue*. Read DMA requests are handled as if they were Write DMA's from the local processor; however, a command buffer is fetched from the *Read Service Queue* pool, and a new job description is posted in the *Network Job List*.

4 Hardware Cost, Latency and Software Evaluation

This section reports on the implementation cost of our FPGA prototype, presents latency figures and evaluates software operations on top of our primitives. First, we report on the total area complexity of the prototype and then we compare plain cache and scratchpad designs against our integrated Cache/Scratchpad and NI. Finally, we illustrate the latency of the primitive operations supported by our NI and present some case studies with software evaluation.

4.1 Design Cost in FPGA Resources

Table 1 presents the hardware cost of the system blocks. The numbers refer to the implementation of the design in a Xilinx Virtex-5 FPGA (XUPV5-LX110T development board) with the back-end tools provided by Xilinx. The most complex block of our NI design is the Outgoing engine which serves jobs from the *Network Job List* and implements a low latency RDMA engine that supports arbitrary byte alignments and sophisticated packet segmentation. The outgoing

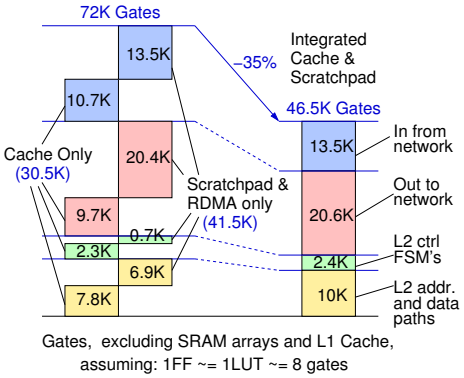


Fig. 6: Comparison of the area complexity for three separate designs: (i) Cache only, (ii) Scratchpad and RDMA-only and (iii) Integrated Cache and Scratchpad.

NI engine costs approximately 40% of the total NI LUTs and 45% of the total NI Flip-Flops. The current total design occupies less that 65% of the available LUTs and Flip-Flops in our FPGA device, however we utilize 90% of the available memory blocks (BRAMs) and thus larger caches cannot be implemented.

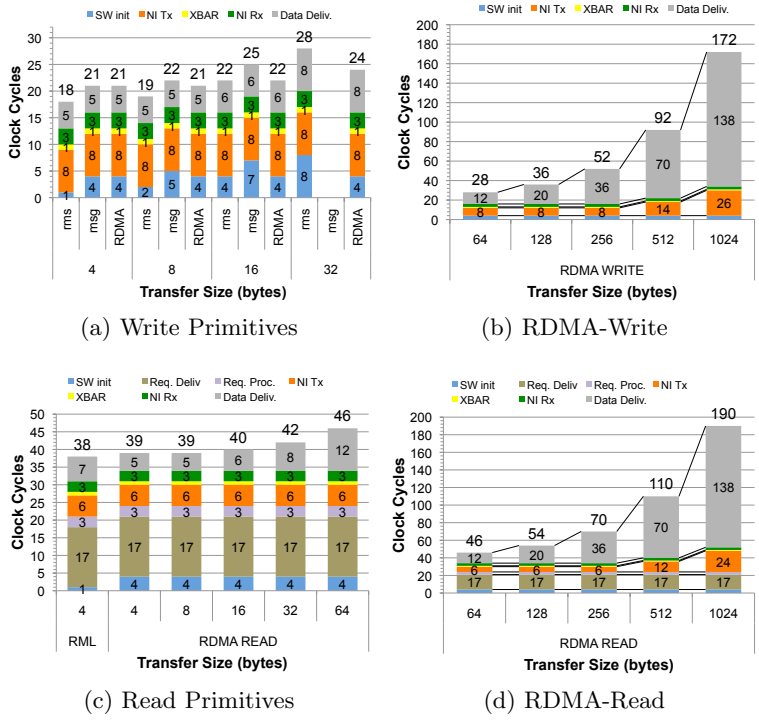
4.2 Area Benefits of Integrated Cache/NI Controller

We have counted and report separately, in Figure 6, the area complexity of three different designs: *(i)* all SRAM operating as cache only, and a cache controller; *(ii)* all SRAM operating as scratchpad only, and a NI providing DMA's; *(iii)* our configurable cache/scratchpad with its integrated NI/cache controller. The cache only design supports one outstanding miss, while serving hits under single miss, and does not support coherence. The scratchpad only design supports 8-byte aligned RDMAs and network packet segmentation.

The area here is reported in gates, to ease comparison, assuming that each LUT and each Flip-Flop is equivalent to 8 gates. The measurements do not include the L1 cache and the memory arrays. As seen, the integrated design *(iii)* has a complexity considerably lower than the sum of the complexities of the two dedicated designs, owing to several circuits being shared between the two functionalities. The circuit sharing is mostly observed on memory block datapath, the outgoing and incoming NI, and economizes 35% in hardware complexity.

4.3 End-to-End Latency

Figure 7(a) presents the latency breakdown of the following primitive NI operations: Remote-Store, Message and RDMA-Write transfers. The SW initiation cost, the NI transmit latency, the crossbar (XBAR) latency and the NI receive latency of every operation are constant under zero network-load conditions – both the outgoing and incoming path implement cut-through. The latency for



(a) Write Primitives (b) RDMA-Write

(c) Read Primitives (d) RDMA-Read

Fig. 7: Remote-Store, Message, RDMA-write, Remote-Load and RDMA-read transfers latency breakdown, as a function of data size (bytes)

the delivery of the packets' payload in the remote memory is commensurate to the size of the transfer.

Remote-Stores of 4-bytes cost 18 cycles and are faster than the equivalent messages and DMAs, since the initiation is implicit – no descriptor has to be posted. Minimum-sized messages and RDMAs of 4-bytes have the same end-to-end latency of 21 clock cycles. Although the RDMA has to read the payload from memory, and implies an extra memory access when compared with the case of a message, the outgoing NI manages to hide this extra latency during the NoC arbitration stage. For transfer sizes larger than 16-bytes RDMA achieves lower latency than remote stores and messages, however RDMA requires the packet payload to be already present in memory, thus is suitable for larger bulk transfers. The latency for large RDMAs is presented in Figure 7(b) which shows that 64-bytes can be delivered remotely in just 28 clock cycles while 512-bytes cost only 92 clock cycles.

The NI transmit path has a latency of 8 clock cycles: 2 of them are attributed to the pipelined path to reach L2, 1 to enqueue a request in the network job-list, 1 for the outgoing NI to process the new request and 4 of them are spent on

the NoC arbitration. The NoC request-grant phase takes 2 clock cycles but the granted network slot starts 2 clock cycles later – during that time the NI hides the latency of reading from memory and preparing packet headers. For transfer sizes that exceed the maximum network packet size, i.e. 256-bytes, and need to be segmented, an extra latency of 6 clock cycles is experienced per segment: 2 clock cycles are spent to recycle a request through the *Network Job List* and 4 clock cycles are spent again in NoC arbitration.

The NI receive path latency has two components: *(i)* the incoming cut-through latency and *(ii)* in-place delivery of packet's data in the memory. The incoming cut-through path has a latency of 3 clock cycles: 2 clock cycles are needed to receive the packet headers and check CRC and 1 clock cycle is needed to inform the incoming DMA engine about a new packet arrival. The incoming DMA engine, that delivers data in-place, needs 2 clock cycles to dequeue the packet headers from the incoming network queues and the remaining latency, until the last word is delivered in memory, is commensurate to the payload size. For 32-byte packets, 4 clock cycles are need to gather a 256-bit chunk and 2 additional clock cycles are needed in order to arbitrate for the tag and memory arrays. The memory arbitration latency is overlapped with the gathering of the next packet words and thus experienced only once per packet.

Figure 7(c,d) illustrates the latency breakdown of primitive remote read operations: Remote-Load and RDMA-Read. Besides the SW initiation cost, all remote read operations have a constant latency of delivering a request to a remote node which is 17 clock cycles – equal to delivering a packet of 8-bytes, contains the destination address for the source node. Thereafter, the request takes 3 clock cycles to be processed by the NI and be converted into an RDMA-write, as if it was initiated locally. The response latency follows the same steps with an RDMA-write and experiences the same latencies. Back at the initiator, the reception of a Remote-Load response takes an extra 2 clock cycles, when compared to and RDMA-Read, since the data should follow the L2 pipeline and be returned to the processor – RDMA-Reads are delivered in the L2/Scratchpad memory. A remote load of 4-bytes costs as low as 38 clock cycles while an RDMA -Read of the same size costs 39 clock cycles. Reading 64-bytes from a remote node costs just 46 clock cycles while reading 512-bytes takes 110 clock cycles.

4.4 Case Studies: Software Use of Hardware Primitives

This subsection focuses on the use of the proposed hardware primitives by software constructs and illustrates some common cases where our primitives find use. Apart from minimizing the latency of data transfers through virtualized low-latency RDMA and remote stores, software can use our primitives to efficiently implement higher level operations such as: (i) Transfer Completion Notification, (ii) Barrier, and (iii) Distributed and Centralized Task/Job Dispatching.

Transfer Completion Notification: We study a common scenario where a producer sends data to a consumer in pre-agreed buffer space that forms a circular queue. The consumer needs to know when all data have arrived and typically

Table 2: Comparison of software-only operations vs. hardware-assisted.

Transfer Completion			
size	clock cycles/iteration		improv.
(bytes)	SW only	HW Cnt.	percent
200	233	206	13%
500	552	449	23%
1000	1084	831	30%
2000	2152	1620	33%

Barrier			
cores	clock cycles/barrier		improv.
#	Lock Based	HW Cnt.	factor
1	111	41	2.7x
2	194	66	2.9x
3	357	78	4.5x
4	574	84	6.8x

Distributed Task Scheduling			
Masters	clock cycles/task		improv.
Workers	Lock Based	HW SRQ	factor
1M - 1W	199	40	4.9x
2M - 1W	152	40	3.8x
3M - 1W	151	40	3.8x

Centralized Task Scheduling			
Masters	clock cycles/task		improv.
Workers	Lock Based	HW MRQ	factor
3M - 1W	232	87	2.6x
2M - 2W	237	44	5.3x
1M - 3W	270	35	7.7x

a software-built protocol manages the low-level details. The use of interrupts for the reception of packets at the consumer is prohibitive due to frequent context-switches (especially for small packets) and thus packet reception is typically triggered by checking a flag in the last word of the packet. The problem becomes harder when out-of-order networks come into picture and when the transfer size exceeds the maximum network packet size. The producer has to squeeze flags in the buffers to be transferred and the consumer needs to poll all these flags before arrival is triggered; additionally data are not contiguous in the buffer space since the flags have been injected. Our proposed solution is the use of *Counters* and the acknowledgment address offered by RDMA operations, Section 2.3. A counter per-buffer can be allocated at the consumer side and the producer can use its address as acknowledgment address when it issues RDMAs.

We measure the performance of these two sketched implementations in the FPGA prototype for a scenario where 10000 buffers are produced and sent with RDMA to a circular queue with 4 buffer slots at the consumer. For the measurements we vary the buffer size using the following values: (i) 200 bytes, (ii) 500 bytes, (iii) 1000 bytes and (iv) 2000 bytes; sizes beyond the maximum network packet size, i.e. 256-bytes, generate multiple RDMA segments. As illustrated in Table 2, the HW counter approach offers up-to 33% improvement over the software-only approach; the performance gains increase with the size of the transfer since the number of RDMA segments increases.

Barrier: It is a very common operation used by parallel programs to synchronize a number of parallel threads/tasks. The typical software implementation, for a few participating threads, involves a lock-protected memory location which is increased when each thread reaches the barrier; the last thread that reaches the barrier wakes-up all other waiting threads. In lieu of atomic instructions on MicroBlaze, we use an external hardware mutex module, provided by Xilinx, that is placed on the memory bus and allows *test-and-set (TAS) like* operations

in a "fast" non-cacheable address space (SRAM). Using the hardware mutex module, we implement a sense-reversing centralized barrier.

The barrier implementation using the *Counter* primitive is straightforward: the counter is initialized with the number of threads (negative value) and each thread sets a local scratchpad address as notification address of the counter (up-to four supported). Upon reaching a barrier, increments to the counter are sent through remote stores. When the counter becomes zero, it triggers automatic notifications to the pre-configured notification addresses. Multiple counters can be chained (counter notifies counters) to create larger wake-up trees and thus support higher number of cores in a scalable manner [9].

We measure and compare in Table 2, the performance of the two implementations in an empty loop with 10000 back-to-back barriers, while varying the number of threads from 1 to 4. The HW counter is up-to 6.8 times faster than the equivalent lock based implementation on 4 cores.

Distributed and Centralized Task Dispatching: Spawning and dispatching tasks/jobs is crucial in parallel and distributed systems, thus we study two cases of task dispatching: *(i)* distributed and *(ii)* centralized. Case *(i)* refers to a set of masters which initiate tasks to specific workers (statically scheduled): each worker maintains a queue where multiple masters may enqueue tasks but only the owning worker may dequeue (many-to-one communication). Case *(ii)* refers to a central pool of tasks (queue) where multiple masters may enqueue and multiple workers may dequeue allowing for dynamic scheduling and load-balancing (many-to-many communication). The typical software implementation of *(i)* requires the masters to acquire a lock in order to enqueue a task and increase the tail pointer, while the worker may dequeue without acquiring a lock. However, in case *(ii)*, where multiple workers dequeue, a lock is also required to guard the head pointer. Our proposed solution for *(i)* is a *Single Reader Queue (SRQ)* per worker and for *(ii)* a central *Multiple Reader Queue (MRQ)*; these primitives offer atomic enqueue and dequeue operations, Section 2.3.

We measure and compare the performance of the software-only vs. hardware assisted implementations is an program where each master spawns 10000 empty tasks. We vary the number of masters and workers accordingly and report the results in Table 2. For case *(i)* the lock based enqueue incurs an overhead which, for 1 master, cannot be amortized by the task size, whereas some of the overhead is overlapped with multiple masters. The SRQ implementation performs up-to 4.9 times faster and the number of masters does not influence the task processing time. In case *(ii)*, the lock contention increases the task processing time when multiple workers serve tasks from the central queue. On the other hand, the MRQ performs very well allowing for up-to 7.7 times faster processing of tasks.

5 Related Work

Configuration of memory blocks has been studied before in the Smart Memories [10] project, but from a VLSI perspective. They demonstrate that using their custom "mats", i.e. memory arrays and reconfigurable logic in the address and data

paths, they are able to form a big variety of memory organizations: single-ported, direct-mapped structures, set-associative, multi-banked, local scratchpad memories or vector/stream register files. The TRIPS prototype [11] also implements memory array reconfiguration, but in very coarse granularity. They organize arrays into memory tiles (MTs), which include an on-chip network (OCN) router. Each MT may be configured as an L2 cache bank or as a scratchpad memory, by sending configuration commands across the OCN to a given MT.

Network interface (NI) placement in the memory hierarchy has been explored in the past. In 90's, the Alewife multiprocessor [12] explored an NI design on the L1 cache bus to exploit its efficiency for both coherent shared memory and message passing traffic. At about the same time, the Flash multiprocessor [13] was designed with the NI on the memory bus for the same purposes. Cost effectiveness of NI placement was evaluated assessing the efficiency of interprocessor communication (IPC) mechanisms. Mukherjee et al. [14] demonstrated highly efficient messaging IPC with a processor caching buffers of a coherent NI, placed on the memory bus. Streamline [15], an L2 cache-based message passing mechanism, is reported as the best performing in applications with regular communication patterns among a large collection of implicit and explicit mechanisms in [16]. Moreover, NI Address Translation was extensively studied in the past to allow user-level access, overcoming operating system overheads [17], and leverage DMA directly from the applications [13].

6 Conclusions and Future Work

The development of our FPGA prototype and the hardware cost analysis of the configurable cache/scratchpad memory with the integrated *Network Interface and Cache Controller* proves the feasibility of our approach and the existence of circuitry that is shared between the network interface and cache controller. Our implementation shows that the merged cache plus scratchpad uses 35 percent less hardware than the two separate systems. Moreover, bringing the NI close to the processor, at L2 level, has significant performance impact in the latency of NI operations: one-way, end-to-end, user-level communication completes within about 20 clock cycles for short transfer sizes. Additionally, the use of our primitives by software constructs offers important performance benefits in a set of case studies. We are working towards merging the NI functionality with more advanced cache features and directory-based coherence.

Acknowledgments

This work was supported by the European Commission in the context of the projects SARC (FP6 IP #27648) and UNiSIX (Marie-Curie #509595). We also thank, for their assistance in designing the architecture and developing the prototype: Dimitris Nikolopoulos, Alex Ramirez, Georgi Gaydadjiev, Spyros Lyberis, Christos Sotiriou, Dimitris Tsaliagos, and Michael Ligerakis.

References

1. R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. In *Proc. 10th Intl. Symposium on HW/SW Codesign (CODES)*, Colorado, 2002.
2. J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Develeopment*, 49(4/5):589–604, 2005.
3. U.J. Kapasi, S. Rixner, W.J. Dally, B. Khailany, J.H. Ahn, P. Mattson, and J.D. Owens. Programmable Stream Processors. *IEEE Computer*, 36(8):54–62, 2003.
4. K. Fatahalian, T.J. Knight, M. Houston, M. Erez, D.R. Horn, L. Leem, J.Y. Park, M. Ren, A. Aiken, W.J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proc. ACM/IEEE Conf. on Supercomputing (SC)*, Florida, 2006.
5. P. Bellens, J.M. Perez, R.M. Badia, and J. Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *Proc. ACM/IEEE Conference on Supercomputing (SC)*, Tampa, Florida, 2006.
6. M. Katevenis. Interprocessor Communication seen as Load-Store Instruction Generalization. In *The Future of Computing, essays in memory of Stamatis Vassiliadis*, Delft, The Netherlands, September 2007.
7. E. Markatos and M. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proc. of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, San Jose, CA USA, 1996.
8. E.A. Brewer, F.T. Chong, L.Tl Liu, S.D. Sharma, and J.D. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, St. Barbara, 1995.
9. S. Kavadias, M. Katevenis, M. Zampetakis, and D.S. Nikolopoulos. On-chip Communication and Synchronization with Cache-Integrated Network Interfaces. In *Proc. ACM Intl. Conf. on Computing Frontiers (CF'10)*, Bertinoro, Italy, 2010.
10. K. Mai, T. Paaske, N. Jayasena, R. Ho, W.J. Dally, and M. Horowitz. Smart Memories: a Modular Reconfigurable Architecture. In *Proc. of the 27th International Symposium on Computer Architecture (ISCA)*, 2000.
11. K. Sankaralingam, R. Nagarajan, R. Mcdonald, R. Desikan, S. Drolia, M.S. Govindan, P. Gratz, D. Gulati, H. Hanson, Changkyu Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S.W. Keckler, and D. Burger. Distributed Microarchitectural Protocols in the TRIPS Prototype Processor. In *Proc. of the IEEE/ACM Intl. Symposium on Microarchitecture (MICRO)*, 2006.
12. J. Kubiatowicz and A. Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proc. of the ACM Intl. Conf. on Supercomputing (ICS)*, Tokyo, 1993.
13. J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. *ACM SIGOPS Oper. Syst. Rev.*, 28(5):38–50, 1994.
14. S. Mukherjee, B. Falsafi, M.D. Hill, and D.A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proc. of the 23rd International Symposium on Computer Architecture (ISCA)*, 1996.
15. G.T. Byrd and B. Delagi. Streamline: Cache-Based Message Passing in Scalable Multiprocessors. In *Proc. of the Intl. Conf. on Parallel Processing (ICPP)*, 1991.
16. M.J. Byrd, G.T. Flynn. Producer-Consumer Communication in Distributed Shared Memory Multiprocessors. *Proc. of the IEEE*, 87(3):456–466, March 1999.
17. R.A.F. Bhoedjang, T. Ruhl, and H.E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, 1998.