Explicit Communication and Synchronization in SARC

Manolis G.H. Katevenis*, Vassilis Papaefstathiou*, Stamatis Kavadias*, Dionisios Pnevmatikatos*, Federico Silla+, Dimitrios S. Nikolopoulos*

*Institute of Computer Science, Foundation for Research and Technology - Hellas (FORTH-ICS)
+Universidad Politecnica de Valencia, Spain
Email: {kateveni,papaef,kavadias,pnevmati,dsn}@ics.forth.gr , fsilla@disca.upv.es

Abstract

SARC merges cache controller and network interface functions by relying on a single hardware primitive: each access checks the tag and the state of the addressed line for possible occurrence of events that may trigger responses like coherence actions, RDMA, synchronization, or configurable event notifications. The fully virtualized and protected user-level API is based on specially marked lines in the scratchpad space that respond as command buffers, counters, or queues. The runtime system maps communication abstractions of the programming model to data transfers among local memories using remote write or read DMA and into task synchronization and scheduling using notifications, counters, and queues. The on-chip network provides efficient communication among these configurable memories, using advanced topologies and routing algorithms, and providing for process variability in NoC links. We simulate benchmark kernels on a full-system simulator to compare speedup and network traffic against cache-only systems with directory-based coherence and prefetchers. Explicit communication provides 10 to 40% higher speedup on 64 cores, and reduces network traffic by factors of 2 to 4, thus economizing on energy and power; lock and barrier latency is reduced by factors of 3 to 5.

1 Explicit Communication and Network Interface Evolution

Interprocessor communication (IPC) is the basis of parallel processing. IPC can be *implicit*, when the addresses supplied by the software do not identify physical data locations or (time of) movement, or it can be *explicit*, when software (the application, or compiler, or runtime system) is able to also indicate physical placement or transfers, besides specifying computation on data. The SARC architecture [1], supports both implicit IPC, through cache coherence, for ease of programming, and explicit IPC, through scratchpad memories and remote store instructions or remote DMA operations, to be used by software whenever possible for achieving scalable performance.

In order to hide IPC latency, when using implicit communication, we need large issue windows in out-of-order-execution processors, or sophisticated data prefetchers, or both. Explicit communication has the potential to better hide IPC latency, in those cases when software knows better than hardware what transfers need to take place and when. *Remote store* instructions, to addresses that indicate proximity to the consumer, when that is known at production time, will transfer data at the earliest possible time; hardware should coalesce writes to adjacent targets into few network packets, and the processor should not wait for the arrival acknowledgments. *Remote direct memory access (RDMA)* is the other method for explicit communication, in cases that require either reads –when the consumer is unknown or unavailable at production time– or multi-word writes –to achieve good coalescence.

Traditional systems viewed networks as *external* (slow) devices, provided DMA in the network interface (NI), and interacted to it through (slow) input/output (I/O) operations. This is inappropriate for modern systems that incorporate networks on-chip (NoC); in them, RDMA must be accessible through a *low-latency, virtualized, user-level* interface, as opposed to system calls. Within the SARC architecture's global virtual address space, explicit communication is based on directly-addressable *scratchpad* local memories.

Address translation provides processors, accelerators, and tasks with controlled, protected access to selected portions of this global space, including (portions of) local and remote scratchpads. Within this scratchpad space, software can allocate special areas (as many as it wishes) that behave as command buffers, counters, or queues, with *event response* capabilities.

Command buffers are used to issue remote (write or read) DMA operations; counters and queues are used for synchronization, including RDMA completion detection, notifications, and waiting for events. Our approach relies on the same basic principle behind cache operation: for each read or write access, check the tag and the state of the addressed line; for certain combinations of state and access type, side-effect actions must be performed –coherence protocol actions, or RDMA, or synchronization, or event responses and notifications.

The contributions of this work are: *(i)* we architect a network interface that leverages on-chip communication potential with hardware support for synchronization and explicit communication, *(ii)* we introduce the mechanism of event-response applied to cache line state and tag bits, and use it to unify the cache controller and the network interface; *(iii)* we offer a brief overview of contributions of the SARC project in the field of NoC architecture (Section 3); and *(iv)* we use full-system simulation to show that remote stores and remote DMAs achieve performance speedup, reduce latency and dramatically reduce network traffic as compared to directory-based cache coherence even with prefetching (Section 4).

2 Unifying Architecture for Implicit and Explicit Communication

This section describes the architecture of our local memory, which can be dynamically *configured* as partly-cache and partly-scratchpad, and also describes the software interface to the hardware mechanisms for explicit communication and synchronization. These are all based on common hardware actions, thus *unifying* the *cache controller* and the *network interface*: for each access, check the state and tag bits of the addressed line, and act accordingly.

We generalize the traditional approach where the processor uses I/O control and status registers to initiate operations and poll for status or wait for notifications. In order to increase parallelism, multiple pending operations are supported, by means of multiple control and status registers. To reduce overhead, these multiple registers are *virtualized*, so as to be accessible in user-mode. We bring these mechanisms close to the processor, into caches, to reduce latency. To allow *parallel* access, we target *private* –as opposed to shared– caches, and integrate our NI mechanisms into *second-level (L2)* –as opposed to L1– caches, in order to provide sufficient scratchpad space for application data without affecting the processor clock (however the ideas are general and independent of that last choice).

2.1 Memory Access Semantics: Cache, Scratchpad, Communication

To integrate cache, scratchpad, and communication/synchronization, we extend memory *access semantics* using *address translation* to identify explicitly managed (scratchpad) address regions, and cache *line state* bits to indicate different access semantics at finer granularity.

Local and remote scratchpad regions can be identified by their physical address provided via address translation. Alternatively, the address translation mechanism can be augmented with a few extra bits that explicitly determine whether an address region contains cacheable or directly-addressed (scratchpad) data[1], as shown in Figure 1. 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 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. Regions marked as local scratchpad in the TLB occupy a set of blocks in the data portion of the memory "way" block. 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

1. In our prototype [2], we use *Address Region Tables (ART)*, instead of the traditional TLB's, in order to support scalable page migration (only local ART's are updated on local migrations –not all TLB's throughout the entire system), as explained in [3]; that issue, however, is orthogonal to what we discuss in this paper.

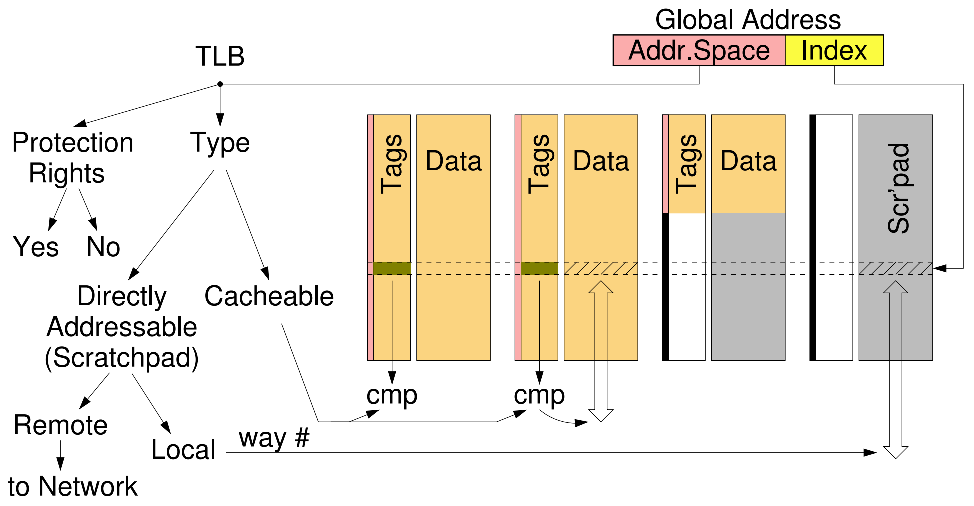


Fig. 1: Memory access flow: the tags of the (configurable) scratchpad areas are not used

granularity, and is used to ignore the actual tag-matching of the hit logic. This 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.

Multiple, *virtualized* communication *control/status "registers"* are implemented in scratchpad region lines. Other than plain scratchpad memory, lines in these regions can be marked, in their state bits, as having *three* types of such special semantics: *(i)* communication *(RDMA or message)* command/status; *(ii) counter*, used for synchronization and notification through atomic increment operations; or *(iii) queue descriptor*, used to atomically multiplex or dispatch information from/to multiple concurrently executing tasks.

The *virtualized* nature of these three types of *event sensitive lines* (ESLs) results as follows: ESLs can be freely allocated in the (virtual) address space of any process. Each process can freely access locally allocated ESLs, in user-mode, independent of, and asynchronously to other processes. ESL access is protected through the normal address translation and protection mechanism (protection is provided for scratchpad regions, e.g. at OS page granularity, not independently per ESL). Virtualization also requires that address arguments passed to ESLs 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 in order to perform these checks. If the operating system should support swapping of a scratchpad out of a certain cache, it has to properly mark and record these special cache lines; to do so, it has to either know their address and type beforehand, or else it can discover them by reading the state and tag bits.

Software can issue a remote DMA operation using 4 store instructions into a command buffer: *(i)* opcode and size; and *(ii)* source, *(iii)* destination, and *(iv)* acknowledgment addresses. Store instructions to scratchpad regions, identified by the processor TLB, should be processed in a pipelined fashion.

2.2 Event Responses

Event responses are hardware synchronization primitives, configurable by software and are provided by counter and queue event sensitive lines (ESLs). On every access, hardware checks the state and tag bits of the addressed line. When that is an ESL and a relevant condition is met, communication *response* is triggered, according to previous software configuration written in the ESL data block.

Counters are intended to provide software notification regarding the completion of an unordered sequence of operations, e.g. multiple transfer reception, or arrivals at a barrier. Counters perform atomic add-on-store. Software can write configuration information, and up to four notification addresses (which may correspond to other ESL's), in their data block. When the counter reaches zero, a pre-configured word is sent to the notification addresses.

Multiple-reader queues (mr-Q) accept asynchronous write (enqueue) and read (dequeue) accesses from any number of processors. Read requests arriving at an empty mr-Q are recorded, waiting until corresponding writes arrive, thus effectively *matching* read and write requests in time. When a write is matched with a read in the mr-Q, a response packet is triggered, with the data of the write sent to the response address of the read. Reads and writes to the mr-Q are buffered in scratchpad memory contiguous to the ESL, forming the queue body. A lock can be implemented with an mr-Q, initialized to contain a single token-word (or a semaphore by initializing with multiple tokens); to acquire the lock, a task reads from the queue and waits for the response packet; to release the lock, the task writes the token back into the queue. Job dispatching can be implemented by writing task descriptors into an mr-Q; whenever processors becomes ready to run a new task, they read (dequeue) an ID from this queue.

2.3 Related Work

User-level network interface access has been extensively researched in the 90's, in order to overcome operating system overheads to communication [4]. Cache integration of the network interface, exploits close coupling with the processor, to reuse its environment and resources for protection and address translation, in order to support communication control and buffering in user space, with fast local access.

Cache-scratchpad configurability has been proposed elsewhere and a few of its variants are supported in some embedded processors. Ranganathan et al. [5] proposed associativity-based partitioning and overlapped wide-tag partitioning of caches for software-managed partitions (among other uses). The cache subsystem in PowerPCs allows locking of cache contents and Intel's Xscale allows per line locking for virtual address regions either backed by main memory or not. Our design generalizes the use of line state for configurable communication initiation and synchronization in addition to locking lines in the cache.

Prior work on fine-grain access control [6] and application specific coherence protocols [7] 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 [8] supported barrier/eureka FSM-based synchronization hardware. Our counter-based barrier support is more general, combines equivalent FSMs with the local memory and is thus easier to virtualize.

Leverich et al. [9] provide a detailed comparison of caching-only versus partitioned cache-scratchpad on-chip memory systems for CMPs, and find that hardware prefetching and cache optimizations eliminate the advantages of the mixed environment. However, they consider communication between on-chip cores and off-chip main memory. By contrast, for on-chip core-to-core communication, RDMA provides significant traffic reduction, which together with event responses and NI cache integration are the focus of our work.

Syncretic adaptive memory (SAM) [10] integrates a stream register file (SRF) with a cache, exploiting compiler mapping of generalized streams. SAM targets a stream processing environment and does not provide support for event handling. Exploiting caches for streaming data in general purpose systems was considered in [11] via hardware support and in [12] via the compiler. Our design integrates equivalent and more scalable mechanisms inside caches than those of [11], providing virtualized RDMA support for efficient bulk transfers.

Our work on explicit communication and synchronization for the SARC architecture includes an FPGA prototype described in [2] and a longer description of the architecture, with performance measurements collected on the FPGA prototype [13].

3 Network-on-Chip Architecture

Network interfaces interact with both the processor and the network. This section briefly reviews the SARC project contributions to the field of networks-on-chip (NoC), in the areas of topology, routing, and process variation tolerance.

Concerning NoC topologies, 2D meshes are the common choice due to their direct mapping to the planar surface of the chip. Nevertheless, other options, like topologies with more than two dimensions, or fat-trees may be very attractive for NoCs. In the first case, multidimensional topologies result in

Parameter	Setting
Cores	1, 2, 4, 8, 16, 32 or 64, in-order UltraSPARC III+ at 2 GHz
L1 I/D Caches	64KB, 2-way associative, 64-byte block, 1 port, 1 clock cycle latency
L2 Caches	private 256KB, 16-way associative, 64-byte blocks, 2-port, 7 clock cycle latency, 32 MSHR, unified, coherent, non-inclusive
Coherence Protocol	MOESI distributed directory, per memory channel, 4 virtual networks, 10 clock cycles latency, up-to 32 directory protocol engines, non-blocking
Data Prefetcher	PC-based stride and tagged, 512-entry history table prefetching degree: 1, 2, 4
Scratchpad	SW based dynamic allocation in L2, block granularity, L1-cacheable
RDMA Controller	SW based dynamic allocation in L2 of command buffers, 64KB max transfer
Remote Stores	two 64-byte coalescing remote store buffers
Network-on-Chip	Concentrated Mesh at 2 GHz, 4 cores per node, 16-byte control packets, 80-byte data packets, 8-byte links, 1 clock cycle link traversal, 5-stage router pipeline 4 virtual networks, 4 VCs per virtual network
DRAM	1GB off-chip DRAM, up-to 8 memory channels, 1 channel per 8 cores, 80ns access time

TABLE 1: Full system simulation configuration

reduced packet latency because of the smaller number of hops required to reach a given destination [14]. Additionally, this effect is more noticeable when combining topologies with more than two dimensions with an increment in the number of cores per switch. On the other hand, the use of fat-trees, combined with the appropriate routing algorithms, provide better performance for medium and large size NoCs [15].

Regarding routing in NoCs, SARC provides efficient and compact implementations of routing algorithms, like the Logic-Based Distributed Routing (LBDR) [16]. Relying on three bits per output port at every switch and a small logic, consisting of a few gates, provides almost the same functionality as table-based routing implementations while requiring much less area, power, and time. Additionally, this routing implementation has been enhanced in order to provide support for splitting the network into different regions and broadcasting packets in them [17].

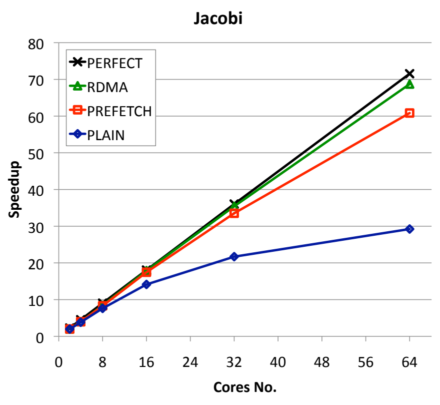
Finally, regarding process variation tolerance, a powerful variability model has been developed [18]: it generates different instances of a chip, for the purpose of evaluating each of them separately, thus getting results on the effects of variability. Starting from a chip layout and the characteristics of a given technology process, the model computes a specific delay for every wire of every link, according to the distribution foreseen by ITRS, some mathematical models, and an initial random seed. The model also provides the maximum operating frequency for each of the routers in the NoC, thus fully characterizing how process variation would affect this particular chip. When applied e.g. to a few thousand of instances of the chip, the model allows the analysis of the effectiveness of architectural approaches to face variability. For example, in a chip-multiprocessor scenario, if the operating system scheduling algorithm used for mapping processes to cores takes into account the variability data from the underlying network, an improvement in application execution time up to 20% can be achieved [19].

4 Performance Evaluation

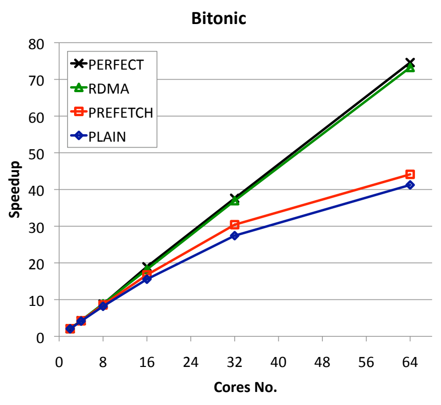
In this section we evaluate our explicit communication and synchronization mechanisms, comparing them against directory-based cache coherence with or without hardware prefetching. First, we measure the performance benefits of explicit communication in a set of popular benchmarks. We proceed to investigate the impact of these mechanisms in on-chip network traffic and power.

4.1 Simulation and Benchmarks

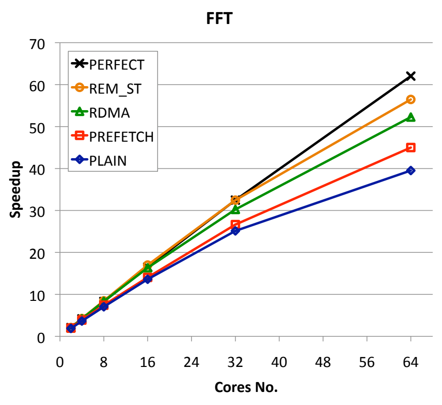
We use SIMICS [20] for full-system functional simulation, and GEMS [21] to implement the timing model for caches, scratchpads, coherence and RDMA controllers. For accurate NoC modeling we use GARNET [22] and measure NoC power with ORION 2.0 [23].



(a) Jacobi



(b) Bitonic



(c) FFT

Fig. 2: Speedup vs core count: Comparing implicit communication with caches and prefetchers versus explicit communication with scratchpads and RDMA.

The configuration parameters of our simulation are listed in Table 1. Cache coherence is based on a GEMS MOESI-directory protocol with distributed directories. The system is configured in 4-processor tiles where each processor core is coupled with private L1 and L2 caches; processor tiles are connected to a single NoC node, forming a 2D concentrated mesh topology. The distributed directories, one per memory channel, are placed on a diagonal-X fashion as proposed in [24], and memory blocks are interleaved across memory channels in cache-block granularity. We further augment caches with strided hardware prefetching optimized with tagging [25]. We have also added our configurable scratchpad memory, ESLs and RDMA engines inside the private L2 caches and segment large RDMAs into multiple maximum sized network packets.

In order to evaluate explicit communication and compare between the two communication models we have chosen three benchmark kernels that exhibit diverse communication patterns: (i) 2D Jacobi, (ii) Bitonic Sort and (iii) FFT. 2D Jacobi is a five-point stencil code where each node communicates with all of its neighboring nodes exhibiting a *nearest-neighbor* pattern. Bitonic sort is a popular sorting kernel where multiple different node pairs exchange data depending on the sorting phase and form the *butterfly* pattern. Finally, we use the Splash-2 FFT kernel to exercise *all-to-all* communication.

We have optimized separately the shared memory and the explicit communication versions of the benchmarks, in order to make fair comparisons. All shared memory implementations are carefully optimized with blocking and the shared arrays are padded appropriately to avoid *false sharing*; MCS [26] locks and barriers are used for synchronization. Porting the shared memory implementations of the benchmarks to use explicit communication, i.e. RDMAs and Remote Stores, and benefit from direct *scratchpad-to-scratchpad* transfers was not a trivial task and required us to fully understand the data exchange patterns. The explicit communication versions of the benchmarks allocate most of the local memory resources as scratchpad and communication buffers and make minimal use cache coherence, mainly to exchange scratchpad buffer pointers during the initialization phases. The latter benchmarks make direct use of the low-level NI communication and synchronization primitives via an optimized lightweight software library, which also provides locks and barriers.

In the evaluations that follow we run all benchmarks with data-sets that fit in the on-chip cache/scratchpad memories in order to isolate and quantify the effects of the communication models in on-chip communication. We keep the data-set sizes fixed and increase the number of cores in order to study the potential of the communication models for fine-grain on-chip communication patterns.

4.2 Benefits from Explicit Communication

For each benchmark we run experiments with up to 64 cores to quantify the effects of explicit communication in execution time. In Figure 2 we present a comparison of the achieved speedups for the following configurations: (i) plain hardware managed caches, (ii) hardware managed caches with strided hardware prefetching, (iii) scratchpad memory with RDMAs and Remote Stores and (iv) a perfect memory system where every memory access costs a single clock cycle. For the measurements of the prefetched version we have experimented with various prefetch degrees and present only the most efficient configuration per benchmark[2].

The speedups[3] measured for Jacobi demonstrate that the RDMA version follows closely the perfect case, i.e. the latency of communication can be perfectly hidden since communication happens only between neighboring NoC nodes. For the hardware prefetched configuration, performance declines on more than 32 cores and cannot follow the perfect case. Using 64 cores for the same problem size, parallelism becomes finer, and communication increase offers the RDMA version a larger advantage that reaches *13% faster* execution than the prefetcher-based version. The reason for this behavior is the excess coherence traffic injected by prefetching: although the prefetcher's efficiency is very high – 96% of the prefetched data are actually used – the associated traffic creates contention in the directories and increases the cache miss latencies. Increasing the prefetching degree further in order to hide the additional latency makes things worse as the traffic is further increased. Moreover, the communication is not restricted to neighboring nodes, as the indirection through the distributed directories forces the cache requests to cross multiple mesh nodes and create additional congestion.

Communication in Bitonic increases with the number of participating processors, thus for small core counts the local sorting phases dominate in the execution time. However, for 64 cores the RDMA version outreaches the prefetcher-based version and results in *40% faster* execution time, ideally following the perfect case. On large core counts communication increases but the amount of exchanged data becomes finer, so they cannot be predicted and prefetched in time. Additionally barrier synchronization time increases on many cores and extra communication is required. On the other hand, our completion notification mechanisms, i.e. counters, trigger local notifications when all data are delivered in place by RDMA, thus saving trips through the NoC.

FFT exhibits an *all-to-all* communication pattern, and for large core counts where computation and communication become fine-grain we observe that no version follows very closely the perfect case, however for 64 cores the RDMA version is *16% faster* than the prefetcher-based version. The bottleneck in the RDMA version is the massive initiation of short RDMA transfers that cannot be amortized. Similarly, the startup overhead (learning period) of the prefetcher cannot be amortized. To alleviate the RDMA initiation cost we have implemented an FFT version with remote stores that incur virtually no cost to communicate with remote nodes. This FFT version is optimized so that the transpose step, the actual communication step, is combined with the FFT computation, through coalesced remote stores. On 64 cores, the remote store version is *9% and 25% faster* than the RDMA and prefetcher-based versions respectively.

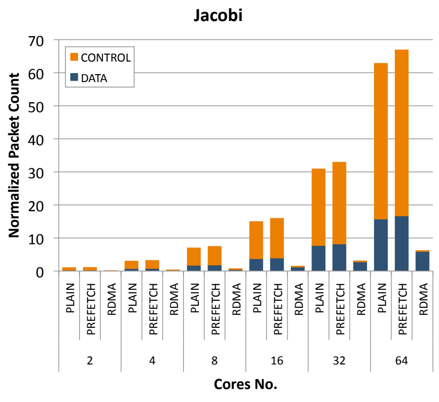
4.3 On-chip traffic

We have collected detailed NoC statistics in order to compare the traffic generated by each communication model. We compare the number of control and data packets generated in each case, and measure the associated transfer volumes in bytes; see Figures 3 and 4 respectively.

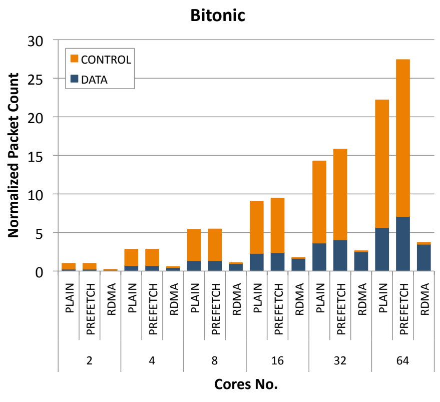
In Jacobi and Bitonic, we observe that the explicit communication using RDMA generates negligible - close to zero- control traffic, and the number of data packets is about half when compared either with plain caches or caches with prefetchers. The reason that caches transmit more data packets is due to the iterative producer-consumer pattern between processor pairs that reuse the same cache lines and force them to ping-pong between nodes – producer requests exclusive ownership and causes invalidation/forwarding at the consumer. In terms of total transferred volume through the NoC, the RDMA version of Jacobi transfers *less than one quarter* of the volume transferred by a plain cache. Similarly, the RDMA version of Bitonic uses *less than half* the NoC volume of a cache.

2. We also used large OS page sizes (up-to 4MB) in order to overcome the prefetcher limitation that does not permit prefetching across page boundaries.

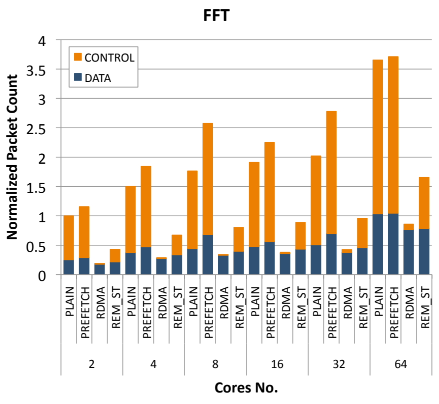
3. Superlinear speedups are an effect of the increased total L1 cache size.



(a) Jacobi

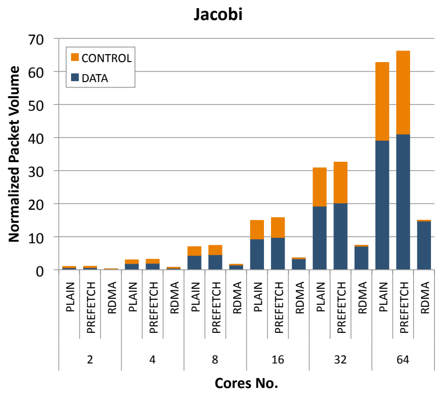


(b) Bitonic

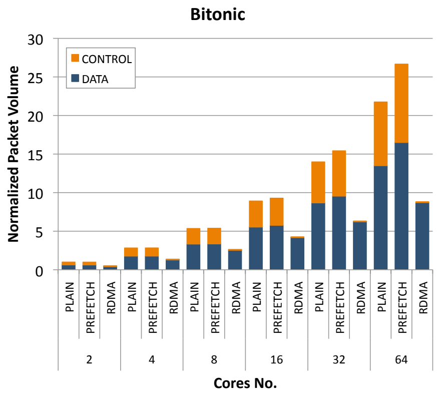


(c) FFT

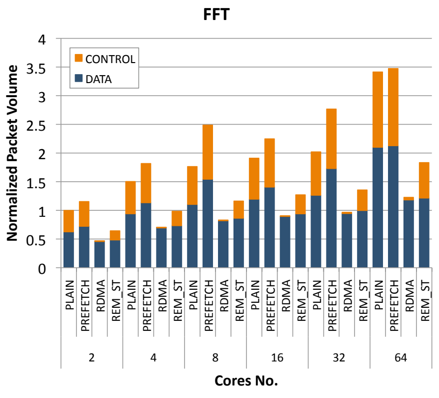
Fig. 3: NoC packet count, normalized to 2-core plain cache total number of packets.



(a) Jacobi



(b) Bitonic



(c) FFT

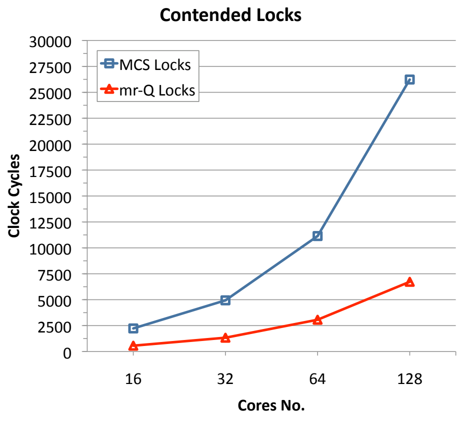
Fig. 4: NoC packet volume (bytes), normalized to 2-core plain cache total volume.

The FFT benchmark for the RDMA version injects far less control packets when compared with the cache version and 30% less data packets: the RDMA version does not need the barrier synchronization required by the shared memory version, but instead uses RDMA completion counters to trigger local notifications when all data arrive. The version with remote stores behaves similarly to RDMA for data packets but requires additional control packets to signal the successful delivery of the remote store packets, i.e. remote store acknowledgments. In terms of total NoC volume, the RDMA and Remote Store versions transfer 60% and 45% less volume than caches.

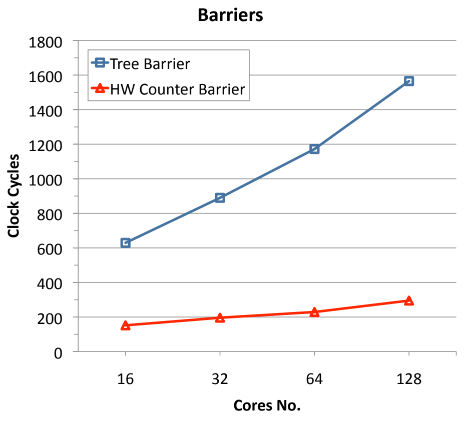
4.4 Explicit Synchronization Performance

We also evaluate the use of *event-responses* for explicit synchronization: Figure 5 compares their performance against the popular MCS locks and barriers [26]. Our locks were implemented with multiple-reader queues and our barriers use counters and notifications. Figure 5(a) shows the average latency of contended lock-unlock pairs of operations. In both implementations requests are queued until the lock is available. The mr-Q based implementation is about *three to four 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 5(b) shows the performance of the two barrier implementations. The counter-based barrier is from $4\times$ faster on 16 cores to $5.3\times$ faster on 128 cores. The main source of increased latency for the tree barrier using coherent variables, is that a barrier requires propagation of signals, for arrival of node or



(a) Locks



(b) Barriers

Fig. 5: Average latency versus number of processors for simulated barriers and locks. MCS locks and Tree-based Barriers use cache coherent communication. mr-Q Locks and Counter Barriers utilize direct scratchpad-to-scratchpad communication and event responses.

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 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.

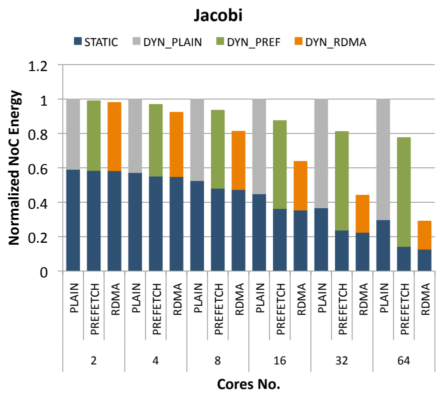
4.5 Network Power

In order to evaluate and compare the impact of explicit communication mechanisms in NoC power, we have carefully collected detailed statistics inside the GARNET [22] NoC models and feed the ORION 2.0 [23] NoC power estimation tool (65nm CMOS technology). We compare the behavior of each communication mechanism using two metrics: (i) Energy consumption and (ii) Power consumption; the results are presented in Figures 6 and 7.

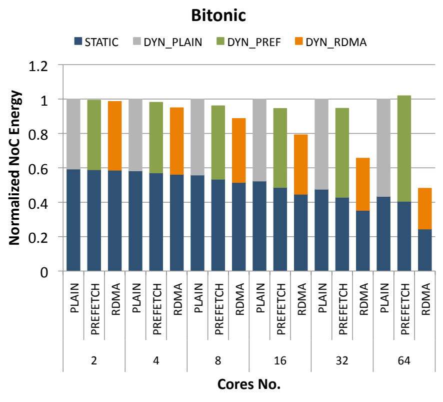
As far as the NoC energy is concerned, the results illustrate that for large core counts, explicit communication achieves significant energy reduction that ranges from 35% to 70% when compared to caches with or without prefetching, Figure 6. The communication intensity of each application greatly influences the dynamic energy consumption, however explicit communication not only is beneficial in all cases but offers additional NoC energy benefits when the applications have high communication demands, e.g. Jacobi. Since explicit communication achieves both lower execution time and lower energy consumption, the energy-delay-product (EDP) metric shows a reduction of 50% to 90% when compared to plain caches and a reduction of 40% to 70% when compared to caches with prefetching. Analyzing the power consumption in Figure 7, shows that the lower energy consumption and EDP of explicit communication is not only an effect of shorter execution time, as it appears for the prefetcher version, but also due to the fact that it generates less NoC packets. The latter effect offers a reduction in the total NoC power that ranges from 15% to 30% when compared with plain caches; in contrast prefetching results in increased NoC power consumption.

5 Conclusions

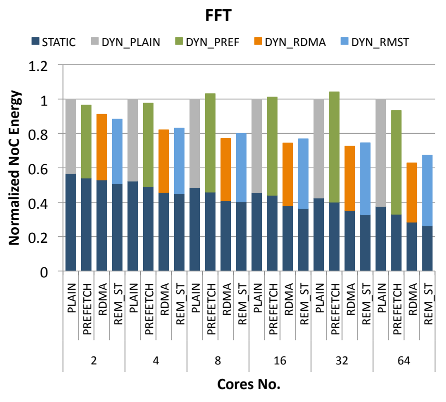
The SARC unified architecture for implicit and explicit communication and synchronization offers concrete advantages in many application cases. It promises ease of programming as the programmer can de-



(a) Jacobi

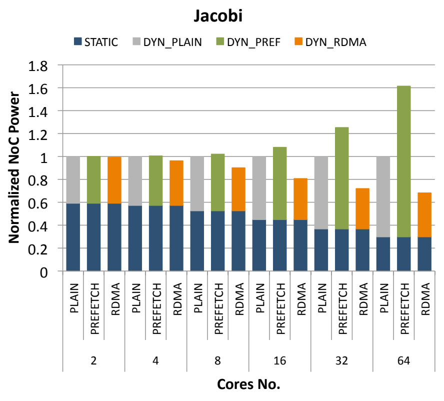


(b) Bitonic

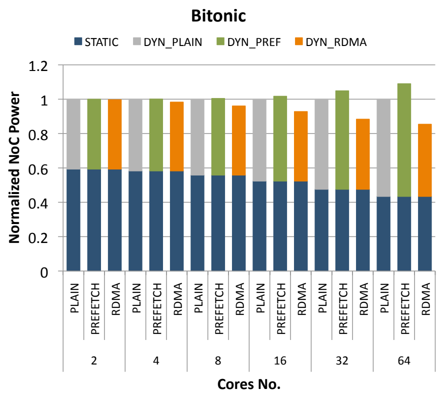


(c) FFT

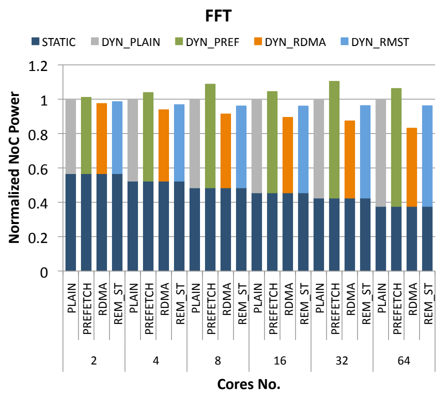
Fig. 6: NoC Energy, normalized to plain cache energy consumption.



(a) Jacobi



(b) Bitonic



(c) FFT

Fig. 7: NoC Power, normalized to plain cache power consumption.

fault to using the caches, with performance and scalability potential of explicit communication – especially for tasks with fine grain parallelism. Combined with our synchronization primitives, explicit communication (RDMAs) drastically reduces control packets and superfluous data transfers, achieving substantial reductions in total network traffic and power. The integration of the NI into the cache/scratchpad controller also reduces communication latency and achieves 10 to 40% faster execution times on 64 cores and reduces network traffic by factors of 2 to 4.

Acknowledgments

This work was supported by the European Commission in the context of the project SARC (FP6 IP #27648) and partially by the projects UNiSIX (Marie-Curie #509595), I-Cores (IRG #224759) and the HiPEAC Network of Excellence (NoE #004408). We also thank, for their assistance in designing the architecture and their collaboration in the SARC project: Alex Ramirez, Georgi Gaydadjiev, Angelos Bilas, George Kalokerinos, George Nikiforos, Dimitris Tsaliagos, Xiaojun Yang, Christoforos Kachris, Giorgos Passas, Spyros Lyberis, Christos Sotiriou, and Michael Ligerakis.

References

[1] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa A. Azevedo, C. Meenderinck, G. Gaydadjiev, C. Ciobanu, S. Isaza, and F. Sanchez. The SARC Architecture. *IEEE Micro*, this issue.

[2] G. Kalokerinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and X. Yang. FPGA implementation of a configurable cache/scratchpad memory with virtualized user-level RDMA capability. In *Proc. IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS2009)*, pages 149–156, July 2009. Extended version to appear in Transactions on HiPEAC.

[3] 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.

[4] R.A.F. Bhoedjang, T. Ruhl, and H.E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, 1998.

[5] P. Ranganathan, S. Adve, and N.P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA'00: Proceedings of the 27th annual International Symposium on Computer Architecture*, pages 214–224, 2000.

[6] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain access control for distributed shared memory. In *ASPLOS-VI: Proceedings of the sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–306, 1994.

[7] B. Falsafi, A. R. Lebeck, Steven K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-specific protocols for user-level shared memory. In *Supercomputing '94: Proceedings of the Conference on Supercomputing*, pages 380–389, 1994.

[8] Steven L. Scott. Synchronization and communication in the t3e multiprocessor. In *ASPLOS-VII: Proceedings of the seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, 1996.

[9] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual International Symposium on Computer Architecture*, pages 358–368, 2007.

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

[11] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally. Architectural support for the stream execution model on general-purpose processors. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 3–12, 2007.

[12] J. Gummaraju, J. Coburn, Yo. Turner, and M. Rosenblum. Streamware: programming general-purpose multicore processors using streams. In *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–307, 2008.

[13] S. Kavadias, M. Katevenis, M. Zampetakis, and D. Nikolopoulos. On-chip Communication and Synchronization Mechanisms with Cache-Integrated Network Interfaces. In *CF'10: Proceedings of the ACM International Conference on Computing Frontiers*, May 2010.

[14] F. Gilabert, S. Medardoni, D. Bertozzi, L. Benini, M. E. Gomezand P. Lopez, and J. Duato. Exploring High-Dimensional Topologies for NoC Design Through an Integrated Analysis and Synthesis Framework. In *2nd IEEE International Symposium on Networks-on-Chip (NOCS)*, April 2008.

[15] C. Gomez, F. Gilabert, M.E. Gomez, P. Lopez, and J. Duato. Beyond Fat-Tree: Unidirectional Load-Balanced Multistage Interconnection Network. *Computer Architecture Letters*, June 2008.

[16] J. Flich, S. Rodrigo, and J. Duato. An Efficient Implementation of Distributed Routing Algorithms for NoCs. In *2nd Annual International Symposium on Network-on-Chip*, 2008.

[17] S. Rodrigo, J. Flich, J. Duato, and M. Hummel. Efficient Unicast and Multicast Support for CMPs. In *41st Annual IEEE/ACM International Symposium on Microarchitecture*, Como, Italy, 2008.

[18] C. Hernandez, F. Silla, and J. Duato. A Methodology for the Characterization of Process Variation in NoC Links. In *Design, Automation and Test in Europe (DATE)*, 2010.

[19] C. Hernandez, A. Roca, F. Silla, J. Flich, and J. Duato. Improving the Performance of GALS-based NoCs in the Presence of Process Variation. In *IEEE International Symposium on Networks-on-Chip, Grenoble, May*, 2010.

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

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

[22] N. Agarwal, T. Krishna, L.S. Peh, and N.K. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *ISPASS'09: Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 33–42, 2009.

[23] A. B. Kahng, B. Li, L.S. Peh, and K. Samadi. Orion 2.0: A fast and accurate noc power and area model for early-stage design space exploration. In *DATE'09: Design, Automation & Test in Europe Conference & Exhibition*, pages 423–428, April 2009.

[24] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 451–461, 2009.

[25] F. Dahlgren and P. Stenström. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(4):385–398, 1996.

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