Direct Communication and Synchronization Mechanisms in Chip Multiprocessors

Stamatis Kavvadias

December 2010

University of Crete
Department of Computer Science

Thesis submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Doctoral Thesis Committee: Manolis Katevenis, Professor (Supervisor)
Angelos Bilas, Associate Professor
Panagiota Fatourou, Assistant Professor
Evangelos Markatos, Professor
Dimitris Nikolopoulos, Associate Professor
Dionisios Pnevmatikatos, Professor
Alex Ramirez, Associate Professor

This work has been conducted at the Computer Architecture and VLSI Systems (CARV) laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH), and has been financially supported by a FORTH-ICS scholarship, including funding by the European Commission.

UNIVERSITY OF CRETE
DEPARTMENT OF COMPUTER SCIENCE

Direct Communication and Synchronization Mechanisms in Chip Multiprocessors

Dissertation submitted by

Stamatis Kavadias

In partial fulfillment of the requirements for
the PhD degree in Computer Science

Author: _____
Stamatis kavadias, University of Crete

Examination Committee: _____
Manolis Katevenis, Professor, University of Crete

_____
Angelos Bilas, Associate Professor, University of Crete

_____
Panagiota Fatouru, Assistant Professor, University of Crete

_____
Evangelos Markatos, Professor, University of Crete

_____
Dimitrios Nikolopoulos, Associate Professor, University of Crete

_____
Dionisios Pnevmatikatos, Professor, Technical University of Crete

_____
Alex Ramirez, Associate Professor, Universitat Politecnica de Catalunya,

Approved by: _____
Angelos Bilas
Chairman of Graduate Studies
Heraklion, November 2010

Acknowledgements

The work reported in this thesis has been conducted at the Computer Architecture and VLSI Systems (CARV) laboratory of the Institute of Computer Science (ICS) of the Foundation for Research and Technology – Hellas (FORTH), and has been financially supported by a FORTH-ICS scholarship, including funding by the European Commission through projects SIVSS (FP6 STREP #002075), Usenix (Marie-Curie #509595), SARC (FP6 IP #27648), ICores (Marie-Curie #224759), and the HiPEAC Network of Excellence (NoE #004408).

I owe a great deal to my advisor, Manolis Katevenis. He was very patient with my difficult character, and persistently positive in his guidance. He taught me to think and write, seeking for precision, and more importantly to look for the bigger picture. I am in his debt for all that and a lot more. I am especially grateful to Dionisios Pnevmatikatos from my advisory comittee, who followed my work on almost weekly basis throughout these six years, was always constructive in apposite ways, and really helped me regain my courage and self-confidence in some crucial points of my study.

I am grateful to my thesis comittee, Dimitris Nikolopoulos, Angelos Bilas, Panagiota Fatourou, Evangelos Markatos, and Alex Ramirez. They have all helped me in different times and with different aspects of the work. Without their suggestions, comments, and guiding lines, this work would have much less to contribute. They all hepled me broaden my perception of computer architecture and I thank them.

This thesis would not have been possible without the continuous and manifold support and the love of my parents. They have followed my ups and downs in these

six years and sensed my difficulties. Although they were staying in a town far from my university, I always felt I was in their thoughts. Despite their distance from an academic background and the process of research, they always found ways to be by my side –thank you.

I owe my deepest gratitude to my friends Nikoleta Koromila, Manolis Mauromanolakis, Antonis Hazirakis, Sofia Kouraki, Elena Sousta, and Ioanna Leonidaki, for always being there for me, for setting up islands of lightheartedness and warm interaction with parties, gatherings, and excursions, and for the numerous times they served my insatiable appetite for discussion on general subjects. I especially want to thank Nikoleta Koromila and Manolis Mauromanolakis for the countless times they have endured my worries, harboured me with comfort, lodged and fed me. I also owe special thanks to Antonis Hazirakis, whose clear and precise judgment helped me stop thinking about quitting after the fifth year, when myself and others started to second-guess the course of my studies.

There are several colleagues who supported and helped me in different ways. Most of all I thank my friend, Vassilis Papaefstathiou, who was always available for deep research discussions, that helped me move forward in my work. George Nikiforos helped me with some of the results, and along with George Kalokerinos implemented and debugged the hardware prototype. Michail Zampetakis also helped me with some of the results. Michael Ligerakis was always prompt in his help with lab-related issues. Thank you all for everything.

Many other friends deserve to be acknowledged. Vaggelis, Thanasis, Nikos, Manolis, Nikos, Alexandros, Minas, Sofia, I am in your debt. There are still other friends and colleagues to thank –please let me thank you all at once.

Abstract

The physical constraints of transistor integration have made chip multiprocessors (CMPs) a necessity, and increasing the number of cores (CPUs) the best approach, yet, for the exploitation of more transistors. Already, the feasible number of cores per chip increases beyond our ability to utilize them for general purposes. Although many important application domains can easily benefit from the use of more cores, scaling, in general, single-application performance with multiprocessing presents a tough milestone for computer science.

The use of per core on-chip memories, managed in software with RDMA, adopted in the IBM Cell processor, has challenged the mainstream approach of using coherent caches for the on-chip memory hierarchy of CMPs. The two architectures have largely different implications for software and disunite researchers for the most suitable approach to multicore exploitation. We demonstrate the combination of the two approaches, with cache-integration of a network interface (NI) for explicit interprocessor communication, and flexible dynamic allocation of on-chip memory to hardware-managed (cache) and software-managed parts.

The network interface architecture combines messages and RDMA-based transfers, with remote load-store access to the software-managed memories, and allows multipath routing in the processor interconnection network. We propose the technique of event responses that efficiently exploits the normal cache access flow for network interface functions, and prototype our combined approach in an FPGA-based multicore system, which shows reasonable logic overhead (less than 20%) in cache datapaths and controllers, for the basic NI functionality.

We also design and implement synchronization mechanisms in the network interface (counters and queues), that take advantage of event responses and exploit the cache tag and data arrays for synchronization state. We propose novel queues, that efficiently support multiple readers, providing hardware lock and job dispatching services, and counters, that enable selective fences for explicit transfers, and can be synthesized to implement barriers in the memory system.

Evaluation of the cache-integrated NI on the hardware prototype, demonstrates the flexibility of exploiting both cacheable and explicitly-managed data, and potential advantages of NI transfer mechanism alternatives. Simulations of up to 128 core CMPs show that our synchronization primitives provide significant benefits for contended locks and barriers, and can improve task scheduling efficiency in the Cilk [1] run-time system, for executions within the scalability limits of our benchmarks.

Contents

List of Figures

List of Tables

1

Introduction

1.1 The Challenge

In the beginning of the first decade of the 21^{st} century, it became apparent that microarchitecture could not derive further performance gains from monolithic CPU designs. The reason was the relative increase of wire delay over denser microarchitectural components with technology advances, which would result in increased cycles per instruction (CPI), and thus in diminishing returns from clock rate scaling [3].

Before the middle of the same decade, the reduced expectation of clock frequency driven performance scaling was coupled with the realization of an even harder problem. The continued frequency and voltage scaling, in addition to increasing dynamic power, was also causing steep exponential increase of transistor sub-threshold leakage power which would become a dominant part of chip power dissipation. As a result, chip manufacturing and packaging technology could not sustain for much longer any strategy to further increase the clock frequency of commodity microprocessors.

The most striking effect of these technology barriers is that any performance increase that the hardware can provide in the future, has to come from increasing the efficiency of either the low-level devices (e.g. in terms of leakage), or the microarchitecture. In other words, we have exhausted for the most part the performance advantage that CMOS device physics could provide via clock and voltage scaling, enabling future chips to operate these devices at the peak power and performance efficiency possible from a physics and manufacturability perspective. In addition, without further clock scaling, essential resources of monolithic out-of-order CPUs have reached the maximum size that can be used effectively (even taking multiple cycles to reach and access them).

Moore's prediction still stands, indicating that semiconductor technology can continue the exponential increase of transistors per chip, so the industry has now turned to the integration of multiple processors in the same chip (also called cores) as a viable way to exploit them. From a manufacturer's perspective, *chip multiprocessors (CMPs)* provide design reuse advantages that help mitigate chip design complexity and thus can keep design time and cost down at reasonable levels.

CMPs also allow trading reduced clock frequencies for more cores and still delivering increased performance, and even managing this tradeoff dynamically with voltage and frequency scaling. For example consider a chip with four cores, each identical to the CPU of a uniprocessor implemented in a previous integration process, that occupies the same area as the uniprocessor. Operating at half the frequency of the uniprocessor, the quad-core chip roughly doubles the processing throughput

of multiple independent jobs, or can provide performance gains from single application parallelization that achieves more than half of linear speedup on four cores. Because the two chips have the same area, they will have roughly the same capacity. Assuming the same voltage as for the uniprocessor, the quad-core chip consumes about half the dynamic power, potentially leaving room for integration of more cores.

In terms of performance, chip multiprocessors have been shown to be more effective in increasing the throughput of server workloads than monolithic CPUs [4], and CMP based supercomputers have raised the Top500 record of performance to 1.75 petaflop/s at the time of this writing. Performance scaling of a single application in a CMP environment depends on the parallelism available in the algorithm, but also on the ability of software and hardware to exploit it efficiently, which can be a complex task. In many cases, considerable effort is required in software to use efficiently the multiple cores for a single application, and hardware must provide primitive mechanisms that allow software to exploit different forms and granularities of parallelism. In effect, chip multiprocessors provide a path for hardware and software to join their forces to increase a computation's efficiency with counterbalanced efforts. It is informally said between computer architects referring to the industry's turn to CMPs that "*The free lunch is over,*" and this is true for both the software and the hardware disciplines.

An essential part of CMP design is the communication architecture that allows cores to interact with each other and with main memory. Only a very limited set of computations can benefit from many cores without frequently requiring such interactions. This dissertation focuses on the communication architecture of chip multiprocessors, and especially the interface of cores to the on-chip network, in consideration of tens or hundreds of cores per chip.

1.2 Towards Many-core Processors

Chip multiprocessors are also called multi-core microprocessors. Lately, the term many-core processors is also used to refer to CMPs with relatively large numbers of cores (at least several tens) that have become feasible as demonstrated by a few experimental chips [5, 6] and actual products [7]. For large numbers of cores, CMPs are likely to utilize a scalable on-chip interconnection network, since a simple bus would introduce large delays for global serialization of interactions. With several tens or hundreds of cores, a multi-stage network is likely to be the dominant choice,

rather than a single switch, to reduce the number of global wires and in favor of localized transfers.

The increasingly distributed environment of many-core processors requires a communication architecture that can support efficiency (i.e. low execution overhead) and programmability in a unified framework. To provide such support, this study targets:

(i) Efficient processor-network interface interaction.

(ii) Enhanced scalability and efficiency of communication and synchronization mechanisms.

(iii) Flexibility in management of communication and data placement.

The central element in architecting efficient and scalable communication is the network interface (NI) that supports inter-processor communication (IPC). To efficiently support different transfer (and computation) granularities, the NI can leverage the on-chip memory resources and promote computation in this same memory. Communication and synchronization off-loading to the network interface can be used to hide and reduce, respectively, their execution overhead. Scalable communication and synchronization mechanisms will be required to efficiently exploit the increasing chip resources. To this end, the NI can provide low latency mechanisms and support for efficient and scalable on-chip networks. Finally, providing in the same architecture the alternative of software controlled data transfer and placement, while also supporting hardware-managed caching and coherence, can be used to exploit the most fitting approach for each computation task.

The following subsections discuss the three main ideas shaping the network interface architecture toward these targets: *sharing communication memory for computation at user-level*, *communication and synchronization avoiding superfluous indirection*, and *supporting enhanced scalability and flexibility*.

1.2.1 Sharing NI Memory for Computation at User-level

Past network interfaces, which were usually placed on the I/O bus, often have provided large private memory, used for intermediate processing of packets and buffering traffic categories until receiver software can handle them. This partitioned arrangement, illustrated in figure 1.1(a), has the disadvantage that both processor and network interface memory may be underutilized: a memory intensive workload would utilize processor memory but not NI memory and the reverse is possible for

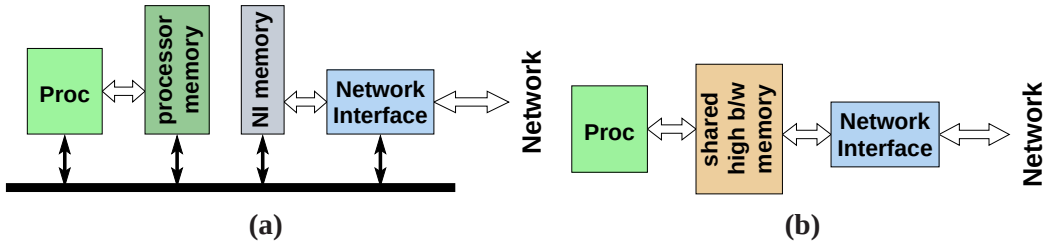


Figure 1.1: Partitioned (a) versus mutually shared (b) processor and network interface memory.

a communication intensive workload. In addition, with this organization, whenever NI memory is used, data copying between the two memories is needed, entailing additional delays and energy wasting.

This thesis advocates an alternative organization, shown in figure 1.1(b), which allows sharing the memory used for communication purposes with the processor, and avoids the inefficiencies of the partitioned one. The shared memory should provide adequate bandwidth for both the processor and the network interface, according to their requirements. This can be achieved through a wide or bank-interleaved memory organization.

In addition, the proposed arrangement is suitable for the aggressive on-chip environment, by directly interfacing processor and network interface through the mutually shared memory. Although user-level sharing of NI memory for computation is also possible with the arrangement of figure 1.1(a), the organization of figure 1.1(b) favors efficient processor-NI interaction and broadens the scope and usefulness of the supported communication mechanisms. Traditionally, caches, optimized over a long period of time for the processor environment, have adopted the latter setting, although their hardwired controllers are not usually addressed as network interfaces, mostly because they are transparent to software.

The organization of figure 1.1(b) can also be realized with scratchpad memories in the processor environment. *Scratchpads* are on-chip memories directly accessible and managed in software, used as temporary storage for code and data. The processor-relative term *local store* is also used for the same purpose, especially when a separately-addressable, software-managed, on-chip memory is provided (or corresponds) to each processor. In contrast to caches, scratchpads require software controlled data placement on-chip, and favor producer-initiated pro-active transfers that deliver data to the consumer without the round-trip required for a read or for write-allocate caching.

Utilizing scratchpad memories, the organization of figure 1.1(b) promotes the

network interface to a *"first class citizen"* among the distributed resources of the chip, coupling it with the processor and the fast memory used for computation. Processor proximity can provide low transfer initiation latency in support of efficient communication mechanisms. Associating the network interface with on-chip memory, allows it to flexibly handle transfers of a few bytes up to several kilobytes. It also allows for processor decoupled (or asynchronous) network interface operation, that can overlap bulk transfers with computation to inexpensively hide latencies, without the need for non-blocking caches. A simple DMA engine can support bulk transfers from and into scratchpad memory, without necessitating processor architecture adaptation to data transfer requirements as in the case of vector and out-of-order processors.

One additional issue has to be addressed regarding scratchpad memories and network interfaces in the processor environment. Low latency access is indispensable for their utility in the on-chip environment of general purpose many-core processors, thus making prohibitive any interaction with the operating system in the common case. In order to support concurrent and protected access by multiple processes, scratchpads and their associated NI must be accessible at user-level.

Protected, user-level access is achievable via memory mapping of resources. In addition, the close coupling of the network interface with the processor can facilitate translation and protection mechanisms in the network interface. Such mechanisms will enable application-space arguments to communication (e.g. virtual addresses for communication endpoints), although circumventing the operating system in the common case. Reversely, receiving transfered data in user-level accessible scratchpad memory, avoids the need for copying between kernel and user memory space.

Caches and user-level accessible scratchpads, utilizing the organization of figure 1.1(b), exploit the advantage that computation occurs "in-place", in the same memory where data are fetched to, without copying. This advantage occurs naturally, because the memory used for computation is also the "communication memory" managed by the cache controller or the network interface.

This thesis advocates a virtualized network interface closely-coupled to the processor, that supports fast local data access and communication initiation at user-level, allows software-controlled data transfer and placement, and exploits NI memory for computation.

1.2.2 Communication and Synchronization Avoiding Unnecessary Indirection

Extending a cache hierarchy to support multiple processor nodes requires that private parts of the hierarchy are kept coherent. Coherent caching over a scalable network-on-chip (NoC) will usually employ a directory-based protocol, which may introduce overheads to inter-processor communication because of the directory indirection required and because of the round-trip nature of caching that employs the usual write-allocate policy.

To the contrary, this thesis advocates the use of *direct communication mechanisms*, to avoid these overheads and exploit the full potential of future many-core processors. Direct communication mechanisms implement *direct transfers*, which are communication operations that utilize network interface controllers only at the source and the destination of a transfer. Since remote read communication operations inherently require a round-trip, they are also direct transfers as long as the only NI controllers involved are at the requestor and at the data source. As an exception, in this dissertation, hardware-assisted copy operations, also addressed as RDMA-copy, that move data from a source to a destination, are considered direct transfers, even when neither the source nor the destination are local to the node initiating the transfer[1].

By definition, direct transfers involve the minimum possible number of device controllers and network traversals for a data transfer. In addition, direct read and write transfers between two nodes of a system allow the most optimized route for each network crossing, which will normally involve the minimum number of network switches. Note that controller access and NoC switch utilization are the primary contributors to both the latency and the energy consumed for an on-chip transfer.

The minimum possible number of involved controllers and the optimized routes for each network crossing of direct transfers, can be contrasted to the case of coherent transfers that require directory indirection. The directory cannot be in the shortest path of any two nodes that need to communicate, entailing overheads, but, more importantly, the directory access itself incurs energy and latency penalties to coherent transfers. Furthermore, in the case of processor-to-processor signals or

[1] Such “triangular” RDMA-copy operations are considered direct transfers because of their similarity to remote read operations: data are transfered from the source to the destination after an initial request. In addition, the involvement of the third controller (the transfer’s initiator) is not superfluous, and any non-speculative communication mechanism implementing such a transfer would involve all three controllers.

when posting operands to a known consumer, producer-initiated direct transfers can provide additional latency benefits over coherent transfers and caching that uses the write-allocate policy, since the latter would require one or two round-trips via the directory for each communicated cache line.

Because of the above, direct transfers could be used to reduce communication overheads, and exposing direct communication mechanisms to software could be employed to enhance communication efficiency in many-core CMPs, provided communication could be orchestrated appropriately for a given application. In addition, the minimum latency advantage of direct transfers can be combined with a favorable position of the network interface, that allows for small communication initiation latency, to construct fast communication mechanisms, appropriate for fine granularity computations in the on-chip environment. Finally, in the context of technology-constrained many-core processors any energy benefits from direct transfers can also be very important.

Synchronization can also become an important source of overhead, as the number of cores per chip increases, especially when many parties have to synchronize. *Synchronization* is the coordination of two or more concurrently executing threads regarding (i) transfer of data ownership between producers and consumers, or (ii) temporary ownership of a shared system resource. In case (i), transfer of ownership is associated with a previous or subsequent data transfer. The transfer can occur between private memories of a producer and a consumer, or between either a producer or a consumer and shared memory. Synchronization, in this case, consists of consumer notification that the producer has completed the transfer or update of data to be consumed, or producer notification that shared data have been consumed and new data can be produced. In case (ii), subsequent owners of the resource are not in general aware of each other, and, perhaps more importantly, the population of contenders may change in arbitrary ways. Coordination can be achieved with mutual exclusion, usually provided with locks.

The case of shared memory and shared data structures is marginal and allows the alternative of viewing a data structure as a shared resource and achieving thread coordination (i.e. synchronization) via mutual exclusion. Alternatively, mutual exclusion can be avoided, exploiting data structure properties and a limited set of supported operations, to provide independent or cooperative progress of concurrent thread access using only atomic operations. The latter is referred to as lock-free or wait-free synchronization [8] and is not addressed in this thesis.

A network interface closely-coupled to the processor can be exploited for efficient synchronization, by decoupling the processor from the synchronization operation in two different ways. First, when the synchronization involves an atomic

operation, this operation can complete in the network interface without involving the processor, which allows for a simpler implementation. Second, when synchronization is associated with notification of a thread about an event, the network interface can automatically initiate a notification signal (a memory write) when a condition is fulfilled. In addition, atomic operations and notifications can be combined to complete complex synchronization operations in the memory system. Because they avoid processor indirection, such mechanisms are called *direct synchronization mechanisms*.

This dissertation extensively studies the design and the implementation of a network interface closely-coupled to the processor, supporting direct communication and synchronization mechanisms intended for large scale CMPs.

1.2.3 Providing Enhanced Flexibility and Scalability

Although cache hierarchies have been the dominant approach to locality management in uniprocessors, cache-design research for up to 16-core CMPs provides two important findings: (i) a shared static NUCA L2 cache has comparable performance to a shared L2/L3 multilevel hierarchy, while a shared dynamic NUCA L2 outperforms the multilevel design [9], and (ii) for best performance, the optimal number of cores sharing L2 banks of a NUCA cache (corresponding to a partitioning of the total L2 cache pool) is small for many applications (2 or 4), and dynamic placement mechanisms in NUCA caches are complex and power consuming [10]. Other technics that identify the appropriate degree of L2 cache sharing based on independent data classes, do not address the requirement for communication [11].

These findings indicate that cache sharing becomes increasingly inappropriate as the number of cores increases, which limits the usefulness of a NUCA cache framework as an alternative to multilevel cache hierarchy coherence. The directory state required for coherence is proportional to the number of processors (unless partial multicasts are used) and the amount of on-chip memory. In addition, cache-coherence is not expected to scale performance-wise beyond some point. Perhaps more importantly, the structure of a coherent cache hierarchy limits the flexibility of tiling and design reuse, complicating a low effort approach to incremental scaling of the number of cores. It is thus very likely, that coherence in large scale CMPs will not be supported throughout the chip, but rather only in “coherent-island” portions of the chip complicating programming.

The use of local stores and direct transfers has challenged the general purpose CMP domain –traditionally based entirely on caches and coherence– with the Cell

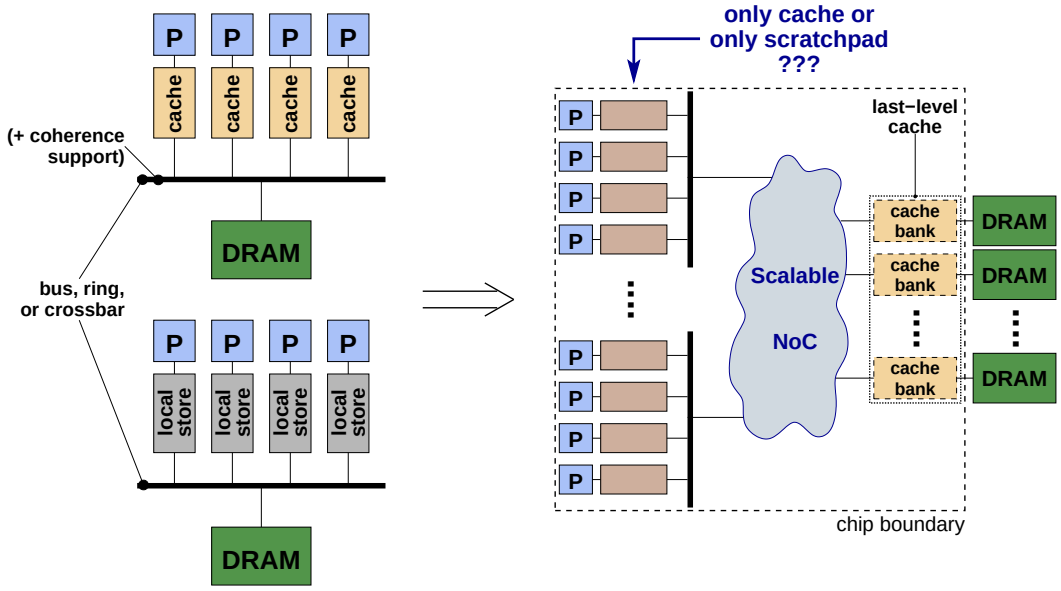


Figure 1.2: From small-scale CMPs to a scalable CMP architecture. On-chip memories can be used either as cache or as scratchpad.

processor [12, 13]. Efficient software communication control has been demonstrated for several codes [14, 15, 16] leveraging scratchpad memories on the Cell processor, and even automated with runtime support in programming models [17, 18, 19].

Figure 1.2 depicts, on the left side, the two baseline contemporary CMP architectures, based either on caches and coherence or on per processor local stores, and, on the right, a possible large-scale CMP architecture[2]. The baseline CMP architectures utilize an elementary NoC, with the cache-based architecture also providing coherence support. The large-scale CMP architecture features a scalable NoC, able to support transfers among the numerous processors, as well as between the processors and main memory. A last-level cache is also shown, that is likely to be used to filter off-chip memory accesses and reduce the effects of the memory wall.

A central issue, in exploiting the baseline contemporary architectures of figure 1.2 in large-scale CMPs, is whether individual processors or processor-clusters

[2]The illustrated architecture follows a dancehall organization regarding the placement of main memory. It is also possible to adopt a distributed memory organization that provides to each processor cluster direct connectivity with a portion of off-chip memory. For the purposes of the discussion in this section, both organizations have similar properties. Moreover, because of the divergence of on- and off-chip access times, the distributed organization is not expected to provide a significant latency benefit for main memory access, while the dancehall organization can enhance bandwidth via fine-grain access interleaving.

should utilized only cache or only scratchpad memory and the corresponding communication mechanisms, or both should be combined in a general purpose way. The former increases diversity and thus will complicate programming. In addition, current experience with the Cell processor indicates that programming using only scratchpad memories is very difficult, as is evident by the numerous scientific publications of achieving just reasonable efficiency with kernels or applications on Cell[3].

Providing to each processor the flexibility of both cache and scratchpad, with their associated communication mechanisms, could mitigate these programmability difficulties, or other that may emerge for the cache-based architecture, as discussed above. The potential problem of this approach is its hardware cost. Flexible division between cache and scratchpad use of the same on-chip memory and cache integration of a network interface, can moderate this cost and explore the similarities of cache and network interface communication mechanisms.

The on-chip network is a primary vehicle in realizing additive performance gains from large-scale CMPs, for all but those computations that largely incorporate independent tasks. As already discussed, many-core processors will possibly utilize scalable on-chip networks, as illustrated in figure 1.2. The basic property of scalable networks, which allows bandwidth scaling with system size, is that they provide multiple paths to each destination. Multipath routing (e.g. adaptive routing [20] or inverse multiplexing [21]) is used to exploit this property and for network load-balance. It can greatly improve network performance, as well as enhance robustness and scalability.

As resources become more distributed with large numbers of cores, locality management can benefit from coarse granularity communication and mechanisms for bulk transfers, to avoid the additional overhead of many small transfers and to hide latency when possible. Such bulk transfers can benefit from the improved network utilization and the achievable bandwidth provided by multipath routing, to improve the computation's efficiency. In addition, to hide the latency of bulk transfers and of access to off-chip main memory, the NI can support many transfers in parallel. To achieve this goal, the state for outstanding transfers required in the implementation must scale reasonably in terms of the required space, access time, and the energy consumed.

This study targets many-core processors with scalable NoCs that support multipath routing, and a network interface with scalable, low overhead mechanisms, suitable for efficient synchronization, as well as fine and coarse grain transfers,that flexibly shares its resources with the processor's cache.

[3]To be fair, one should attribute some part of Cell's inefficiency to the poor scalar performance of its vector synergistic processors.

1.3 Architectural Support for Many-core Processors

Exploitation of the large number of cores that will be available in future CMPs, will critically depend on the performance of the communication architecture to reduce communication overheads, and locality to avoid these overheads when possible. In addition, cooperation of the multiple cores will require efficient synchronization and low-latency transfers to condense scheduling and synchronization overheads. These prerequisites for many-core exploitation will become more important especially when a workload exposes insufficient parallelism to hardware.

This thesis addresses these requirements in two ways. First, it proposes architectural support for an efficient and flexible communication and synchronization infrastructure. Second, it employs a selection of basic mechanisms, that have been proposed in the past, and adopts them appropriately for the on-chip environment and the proposed communication architecture. The basic mechanisms employed are discussed first. The following subsections summarize the novel architectural enhancements proposed for many-core CMPs.

Extension of the coherent address space of contemporary mainstream CMPs is employed, in support of direct communication over a non-coherent part. The *non-coherent address space* extension will accommodate on-chip and per processor scratchpad memories that provide a comfortable area for application data. This will allow low-latency communication and synchronization, as well as direct transfers of variable size. Furthermore, scratchpad memory occupying part of the shared address space, provides a straightforward path to its user-level access, by exploiting common translation and protection mechanisms as for off-chip memory.

This is more general than the approach adopted in the Cell processor, where processors other than the control and high-performance PowerPC core are not allowed to access directly remote scratchpads (i.e scratchpads of other processors); only the control processor and the DMA engines of auxiliary processors may have direct access to the whole shared address space. There are two important consequences of this restriction: (i) auxiliary processors are forced to compute only on local data and also store results locally, and (ii) auxiliary processors cannot directly access the program's data structures, and need to use the DMA engine to copy data in and out of their local scratchpad[4].

A set of scalable and efficient direct communication mechanisms is also employed. The communication architecture introduced, supports remote loads and

[4]Furthermore, Cell's implementation does not allow auxiliary processors to initiate read DMA transfers from remote scratchpads (read DMA is allowed only from main memory), but this is not related to the use of the address space.

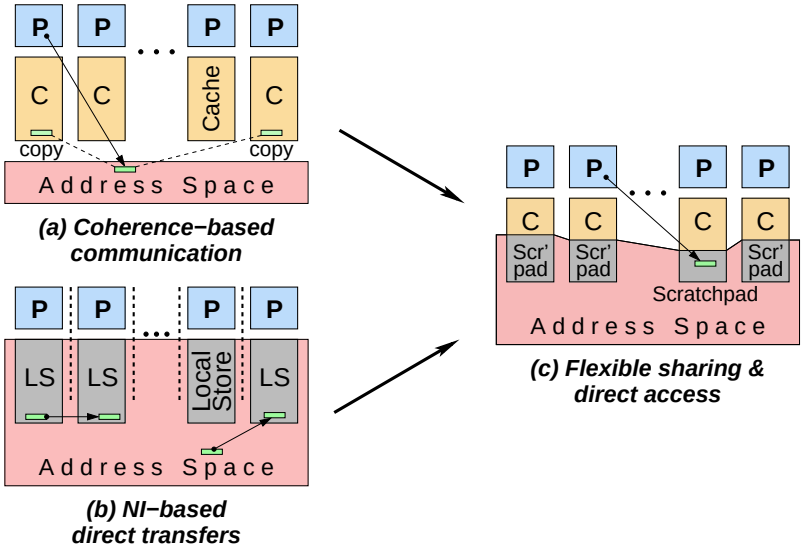


Figure 1.3: Use of the address space by coherence-based indirect communication providing local block copies on demand (a), and extension of shared address space for scratchpad memories that enables direct transfers (b).

stores, messages, and remote DMA (RDMA) with copy semantics to and from non-coherent parts of the address space (RDMA-copy). These may include on-chip scratchpad memories and non-cacheable off-chip memory. Although the interaction between coherent and non-coherent parts of the address space is an important part of the design space exploration, it is not addressed in this thesis. The coherence extensions required for this interaction are studied elsewhere [22].

1.3.1 Cache Integration of a Network Interface

The integration of a network interface at the top levels of the cache hierarchy is proposed. In this context, the term “network interface” refers to direct communication and synchronization operations that are exposed to software. The introduced cache-integrated NI supports efficient sharing of on-chip memory resources between cache and scratchpad use, at cache-line granularity and according to the needs of the application. This allows both cacheable coherent access to parts of the application’s data, processors as well as direct non-coherent access to other parts. Scratchpad memory can be used as NI staging area for direct data transfers, that is directly accessible from the processor at user-level, and thus does not call for data copying.

Consolidation of the two contemporary baseline architectures, with cache-

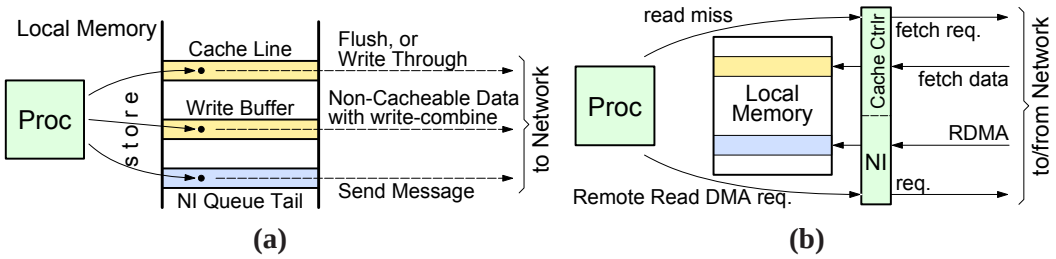


Figure 1.4: Similar cache and network interface mechanisms for writing (a) and reading (b) remotely.

integration of a network interface, is illustrated in figure 1.3. In architecture (a) of the figure, the coherent address space is accessed through locally cached block copies, and communication is based on hardware-managed coherence mechanisms. In architecture (b), the non-coherent address space extends on-chip to include per processor local stores. The non-coherent address space supports the naming of sources and destinations for direct transfers that always utilize the NI. In architecture (c), a flexible mix of coherent and non-coherent address spaces, facilitates partial cache use as scratchprocessorspad, with support for both types of communication mechanisms, and also allows direct processor access to remote scratchpads.

The communication architecture introduced, provisions for an extension of the address space to accommodate per processor scratchpad memories, by exploiting addresses to derive node ID information. Scratchpads are realized inside processor-local caches and are supported by cache-integrated network interfaces[5]. This arrangement allows parallel use of both implicit and explicit communication, depending on software needs, to achieve the best of the two worlds.

In addition, cache-integration of the network interface allows efficient sharing of functionality between them, since the cache already provides a network interface (although a hardwired one). Cache block write-back or flushing can share functionality with operations like sending a message or storing remotely with write combining, as shown in figure 1.4(a). Similarly, a load-miss can share functionality with a read DMA operation since they both generate and send a request, as depicted in figure 1.4(b). Furthermore, the cache-integrated network interface introduced, effectively exploits cache data storage to accommodate scratchpad data or the body of network interface queues (see next subsection), and cache tag storage to keep NI

[5]Regarding the semantics of direct communication and synchronization mechanisms and the address space extention, the use of cache memory for scratchpad and associated integration of cache controller and network interface, is only an implementation approach. Scratchpads with their network interface could equivalently be implemented independently of caches.

meta-data.

Cache-integrated network interfaces do not restrict the quest for efficiency to utilize only caches or only scratchpad memories. Instead, they enable software exploration of the tradeoffs between coherent and direct transfers, without dismissing either. From an implementation perspective, cache-integrations of the network interface effectively shares on-chip resources avoiding the inefficiency of partitioned cache and scratchpad storage with independent controllers.

Supporting both implicit and explicit communication allows software (i.e. programmer, or compiler, or run-time system) to optimize data transfer and placement using efficient explicit communication mechanisms on a best effort way (i.e. when it is feasible or does not require excessive work). In addition, cache-integrated NIs share on-chip memory for cache as well as scratchpad and NI use, and leverage cache control and datapath for the network interface.

1.3.2 Direct Synchronization Support

Direct synchronization mechanisms are proposed, which provide atomic operations and automated notifications, and are implemented in the network interface, avoiding the need for interaction with the processor. The use of direct transfers for notifications avoids the round-trip introduced for signaling with coherence-based communication. In addition, the proposed mechanisms can be combined under software control to provide complex synchronization operations like barriers, locks, and job dispatching, with all the synchronization required taking place in the memory system.

Counter synchronization primitives are proposed, that support atomic addition of a stored value to the contents of the counter (called an *add-on-store* henceforth). When a counter becomes zero, it triggers the transmission of up to four notifications (single-word writes) toward addresses configured in advance by software. In addition, software configures the notification value and a reset value for the counter. Counters can count local and remote write events and automatically send notifications at some expected total count. Thus, they can be used to detect the completion of multiple transfers, accumulating the acknowledgements and providing preconfigured notifications. Similarly, they can detect the completion of multi-packet RDMA transfers, even over an unorder network. In addition, multiple counters can be combined to construct fast hierarchical barriers.

Queue synchronization primitives are also introduced for multi-party synchronization. Two variants are proposed: *single-reader queues (sr-Q)* and *multiple-*

reader queues (mr-Q). Queues can efficiently multiplex multiple asynchronous senders, providing atomic increment of the tail pointer on writes. In single-reader queues, the head pointer is controlled by the local processor. In multiple-reader queues, the head pointer is handled like the tail but reacts to read requests. Multiple-reader queues implement a pair of "overlapped" queues, one for reads and one for writes. The response of a multiple-reader queue to a read request can be delayed, because reads are buffered until a matching write arrives. Multiple-reader queues provide sharing of the queue space by multiple receivers (dequeuers) and can enable efficient lock and job-dispatching services.

1.4 Contributions

This thesis deals to a large extent with the design of NI-cache integration and the resulting communication architecture, as well as its efficient implementation. In addition, the design is evaluated both on an actual hardware prototype, implemented in the Computer Architecture and VLSI Systems (CARV) laboratory of the Institute of Computer Science of the Foundation for Research and Technology –Hellas (FORTH-ICS), and with simulations. The contributions of this dissertation are:

- Design of a cache-integrated network interface for direct inter-processor communication in large-scale CMPs. The cache-integrated NI provides scalable communication mechanisms, including RDMA-copy in a shared address space, supports multipath routing, and efficiently shares on-chip resources for network interface functions. The NI requires less than 20% logic increase over a cache-only design for the integrated controllers and datapath of its basic configuration, plus one or two state bits per cache-line (section 2.4 and chapter 3).
- Introduction of event responses, as a general technique to exploit the normal cache operation and access flow, in integration of software-configurable communication and synchronization mechanisms, which allow the efficient cache-integration of the network interface. Furthermore, the design of efficient communication initiation, transfer completion detection counters, single- and multiple-reader queues, that are confined to the memory resources of the cache (chapter 3).
- Evaluation of on-chip communication mechanisms on the hardware prototype, as well as simulation-based evaluation of the use of NI synchronization primitives for lock and barrier synchronization and for task scheduling using

the Cilk run-time system (chapter 4).

2

Communication and Synchronization Mechanisms

2.1 Background

Henry and Joerg [23] have categorized network interfaces in four broad groups:

(i) OS-level DMA-based Interfaces,

(ii) User-level Memory-Mapped Interfaces,

(iii) User-level Register-Mapped Interfaces.

(iv) Hardwired Interfaces.

Although DMA had only been used in network interfaces that required OS intervention at the time, subsequent research efforts (e.g. [24, 25]) demonstrated its feasibility at user-level. NIs accessible only via the operating system are currently inappropriate for efficient on-chip communication, because of OS invocation overheads. Furthermore, if OS invocation overheads are mitigated, there is no significance in the distinction of this type of NI. In any case, this type of NIs does not constitute a separate category for an on-chip environment. On the other extreme, hardwired NIs, that are transparent to software and do not allow program control over sending or receiving communication and its sementics, are utilized for coherent shared memory interactions and in some dataflow machines (e.g. Monsoon [26]).

The two types of user-level NIs, as adopted for on-chip inter-processor communication, will be discussed in detail in section 2.3. Since the place where explicit communication mechanisms are implemented is traditionally called “network interface”, in the following, when not stated more specifically, the term will refer to NIs that provide support for explicit transfers.

In the following subsections, first we discuss implicit and explicit communication, which cache-integrated network interfaces enable for interprocessor communication. Then we review RDMA and its relation to networks with multipath routing, including previous research of the author [27, 28]. Finally, we refer to the lessons from a connection-oriented approach to designing the network interface [29], within the same project and during the early stage of this dissertation.

2.1.1 Implicit versus Explicit Communication

Since the early days of parallel computing in the 80’s, and also later in the the 90’s, constant debate has persisted between the two dominant programming models, shared memory and message passing. The same debate reappears in the context of CMPs, in a different form, focusing around the two dominant approaches for

software to express communication: implicit and explicit. This twist of the original shared memory–message passing opposition in the context of CMPs, emphasizes the importance of communication for the multiple instruction stream parallelism they provide.

Explicit communication refers to data movement stated explicitly in software, including its source and destination. This depends upon a naming mechanism for communication end-points, and can be supported directly in hardware either via processor instructions (e.g. for messaging), or by providing programmable communication engines per compute node.

Explicit communication forces the user program to specify communication and manage the placement of data. In CMPs, hardware naming mechanisms for communication endpoints can be provided, by identifying the target thread, to support short message or operand exchange via registers or per processor queues, or by specifying parts of a global address space as local to a thread[1], using scratchpad memories. Explicit communication mechanisms can provide direct transfers that each require minimum latency and energy, including transfers from and to main memory, as discussed in subsection 1.2.2.

On the contrary, implicit communication does not provide a mechanism to identify the source and destination of communication. In effect, all communication required is done through shared memory, without the software providing any location information. Software can only indirectly represent communication with consumer copy operations, or with some data "partitioning" and different threads accessing separate partitions in each computation phase (i.e. coordinated data access). Hardware support for implicit communication is commonly provided through the use of coherent caches and hardware prefetchers, that serve as communication assistants, transparent to software. In addition, hardware architectures that favor uniform access of shared memory, like dancehall architectures or fine-grain interleaving of shared memory addresses, can render explicit communication useless, making implicit communication the de facto choice.

Implicit communication relies on hardware for data movement and placement[2]. In many-core processors, the use of caches will necessitate directory-based coher-

[1] The notion of a *thread* is left intentionally ambiguous. Whether it refers to a hardware thread or a software thread has to do with the virtualization of the naming mechanism and not the mechanism itself. Virtualization is addressed in subsection 2.3.3.

[2] The case of software shared memory, implemented on top of a message passing system, can be viewed as a hybrid, which provides implicit communication at the application level, using explicit mechanisms at the OS level. In this case, data placement is managed in OS software. Other hybrid schemes include software-only coherence protocols [30] and, in general, software-extended directory architectures [31].

ence for implicit inter-processor communication. With caches, communication is consumer-initiated in most cases, since usually a write-allocate policy is employed. Because hardware has to manage data placement, migration, and replication, the directory has to be consulted and updated for all transfers. This introduces indirect transfers and additional communication to keep caches coherent, that increase latency and energy per transfer.

Implicit communication provides the illusion that hardware does exactly what the program instructs, but in fact it hides when and if communication occurs. As a result, algorithmic properties are blurred because of implicit communication that is difficult to manage in the algorithm. In addition, current implementations of implicit mechanisms do not allow software hints about data placement and locality management in general, and in effect, they only allow indirect program control over the physical location of data, and restrict its algorithmic administration.

Conversely, management of locality in the limited on-chip space can be far more difficult and a significant overhead when done in software, using explicit communication. Nevertheless, in cases where parts of the data or the computation are amenable to explicit communication, its use can reduce the associated overheads and thus improve the computation's efficiency.

Finally, it must be noted that implicit communication allows transparent migration mechanisms, which enable "in-place" access of shared data structures, without need for management of partial copies or pointer modification in software. Although access to shared data structures usually require mutual exclusion, the alternative requires thread-private memories and partial data copies that need to be managed explicitly, in software (see subsection 2.3.6 for a related discussion).

2.1.2 RDMA and Multipath Routing

Remote DMA is an explicit communication mechanism, widely accepted as appropriate for high performance inter-processor communication (IPC), in multiprocessors other than those based on cache-coherence. Implemented in a shared global address space as RDMA-copy, which is advocated here, it offers a generalization of load and store operations for arbitrary size[3] memory-to-memory transfers. Contrary to simple RDMA that implements get and put operations, requiring that either the source or the destination address is local, RDMA-copy supports global source and destination addresses.

[3]For RDMA-copy to operate with virtual source and destination addresses, a TLB in the network interface may be required to allow a size that results in page boundary crossing.

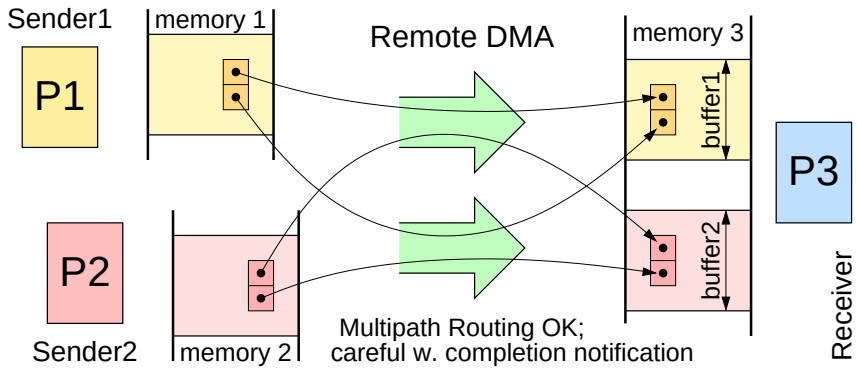


Figure 2.1: One-to-One Communication using RDMA: single writer per receive buffer.

In contrast to register-to-register transfers, RDMA allows varying the transfer granularity beyond the size of the processor's register file. Relative to the delivery of data into receive queues, that may need to be copied into program data structures, it has the advantage of zero-copy, by directly delivering data "in-place". Compared to the copying of data via load-store instructions, it has the advantage of asynchrony, thus allowing communication to overlap with computation. Unlike implicit communication through cache coherence, it can deliver data to the receiver before the receiver asks for them, thus eliminating read-miss latency. Finally, relative to the cases of successful prefetching in caches, since RDMA-copy implements direct transfers, it uses much fewer packets to perform the transfer, thus economizing on energy.

An alternative to RDMA can be provided with processor block-copy instructions, assuming that the cost of temporarily wasting a processor or hardware thread is diminishing. In the author's perspective, RDMA can be more bandwidth efficient, exploiting a wide scratchpad memory organization instead of the processor register file, because blocks of memory can be transferred instead of smaller words that would be of interest to a processor's program. Similarly, with an RDMA engine it is easier to support the overlap (or pipelining) of multiple concurrent transfers, because only transfer state needs to be managed, independently of the program's computation state. Further, RDMA allows processor computation to continue undisturbed, and provides an easy-to-scale implementation for the above functionality as will be described in chapter 3, demonstrated by a hardware prototype. It should be noted that block-copy instructions would not be allowed under sequential consistency, similarly to RDMA-copy operations, because of the inefficiency of placing them in some sequential order.

Figure 2.1 illustrates a typical use of write RDMA with multiple parallel transfers in progress. For multiple senders (P1 and P2), to be sending to a same receiver

(P3), the receiver must have set up separate memory areas where each transfer is taking place. To deliver a large block of data to a remote destination, RDMA segments the transfer to multiple packets, that carry their own destination address each. Owing to this, RDMA works well even if the network uses adaptive or other multipath routing, also shown in figure 2.1, which may cause packets to arrive out-of-order. Delivering packets in-place, RDMA obviates the need for packet resequencing and provides the opportunity to exploit multipath routing that drastically improves network performance.

Resequencing [32, 33] is difficult to implement across a network because, in the general case, it requires non-scalable receive-side buffering. Specifically, previous research of the author and others [27, 28] in the CARV laboratory of FORTH-ICS indicates that, in the general case, the buffer space required for resequencing is proportional to the number of senders, the number of possible paths in the network, and the amount of buffering in a network path. It should be possible to reduce this amount by the last factor with inverse multiplexing [21], but balanced per path and per destination traffic splitting is required at the senders in this case. To the best of the author's knowledge, the only low-cost solution for resequencing is to use end-to-end flow control, which throttles the possible outstanding packets, and causes bandwidth to diminish with increasing round-trip time.

The compatibility of RDMA with out-of-order arrivals over an unordered network, does not solve the problem of detecting transfer completion at the receiver. Completion detection can not be made based on when the "last" word has been written for an RDMA transfer, as would be the case with an ordered interconnect. In [27, 28], resequencing of packet headers was implemented in combination with RDMA to detect transfer completion, and proved non-scalable. Instead, assuming the network never creates duplicates of packets, one can count the number of arriving bytes for each transfer, and comparing them to the number of expected bytes. In following subsections we propose an elegant mechanism that exploits this idea, and provides transfer completion notification in the absence of network ordering.

2.1.3 Lessons from a Connection-based Design

In the context of bringing the network interface closer to the processor and providing scalable communication support, an initial connection-oriented approach was investigated by Michael Papamichael, for his MSc thesis [29] at the University of Crete and the CARV laboratory of FORTH-ICS, within the same project and during the early stage of this dissertation. The conclusions drawn from this work, that discouraged the pursue of a connection-oriented design, are discussed here.

The Papamichael network interface supported user-level access, low overhead and low latency communication mechanisms, and aimed at their highly scalable implementation. Messaging and remote DMA were selected as the communication mechanisms, and message queues as the appropriate synchronization primitive, based on perceived programmer convenience, previous research, and wide use of these mechanisms [34, 35, 36, 37, 38, 39].

Connections would encapsulate the state required for routing to a communication peer and provide a user-level interface to hardware-stored system state, via connection identifiers. A connection table was designed to store all state required for communication that is frequent or active in the current phase of a program, with different parts of this state accessible at user and system levels, and a part accessible only by hardware. These would allow low overhead transfer initiation and protected user-level access.

A connection included separate incoming and outgoing queues and DMA buffers, and provided user-level access to a connection table entry in NI memory. Communication was initiated by writing descriptors to the queue associated with the connection's table entry, which was mapped in the address space of a process at connection establishment. Two types of connections where supported: one-to-one and many-to-many. A one-to-one connection allowed communication and provisioned resources for queue space and RDMA buffers pertaining to a single pair of threads. A many-to-many connection allowed a thread to communicate with many others economizing on NI resources, but introduced higher communication initiation overhead.

Several problems were identified with the connection-oriented approach. The connection table required a lot of NI storage that could not scale to large numbers of connections; yet, to force a scalable solution, it was decided to keep it in NI memory and be of variable size. For the same reason, and because it would be large, direct mapping was chosen for the connection table, instead of fully associate access. The straightforward indexing with the connection ID created problems for thread migration.

Thread migration was problematic in itself, since it required tearing down all connections and setting up new, with all involved nodes. Because each connection ID was tied to a specific index of the connection table, that was known to the application, the old connection ID should be free at the targeted destination node for migration.

Furthermore, connection grouping in many-to-many connections required an extra ID for the sender in addition to the connection ID. Actually, many-to-many

connections forced an additional process group ID for protection and were also creating a problem with destination routing information that was not addressed. Instead, the process group ID and the connection ID of the destination were placed in the network packet (this was probably a first notion of progressive address translation discussed in chapter 3). Finally, the user-level access scheme used, required a whole page to be mapped to the virtual space of a process, to access a few entries of the connection table, which was considered wasteful.

In conclusion, accessing the network interface through a table creates problems with table space and indexing. In addition, there is inefficiency in accessing such NI control state at user-level, over virtual memory, because of the reasonably large protection granularity provided by access control hardware like TLBs.

With a scalable number of destinations such connection state can easily increase to the point that it cannot be managed efficiently. In fact, it seems that the increase is because of the state required for routing to the scalable number of destinations. One-to-one connections, that would encapsulate destination routing information, require non-scalable per node state, and with connection grouping, routing information lookup at transfer initiation is difficult to avoid.

Finally, migration is hard to handle in presence of connections, because connections represent the potential for communication, and migration represents the movement of sources or destinations[4]. Routing to a virtualized destination like a peer process or software thread in presence of migrations is difficult. It requires that setting up and dismissing a path to some destination is as lightweight as possible. This can be supported by a mechanism that gracefully completes the task (e.g lazily or on demand), or by providing an infrastructure that can adapt to the movement of migrating destinations. The importand question here is how often do migrations occur, which, generally speaking, should depend on the granularity of the moving entity, and is related to locality.

2.2 Producer-Consumer Communication Patterns and Mechanisms

This section provides a qualitative comparison of communication and synchronization mechanisms that have been proposed in the literature for implicit and for explicit communication, in the context of producer-consumer interaction. For implicit

[4] A similar difficulty appears with the handling of page migration in presence of address translation.

mechanisms, we consider normal coherent loads or stores and prefetching. For explicit mechanisms, we consider coherence-based write-send [40], remote scratchpad stores, short messages, and simple RDMA. Write-send represents data-forwarding mechanisms, among a few that have been proposed in the literature [41, 42, 40, 43]. Although these are explicit communication mechanisms, they are not implemented with direct transfers advocated in this thesis[5]. A short message, or simply a message in the following, refers to a small transfer that is delivered as a single atomic unit at its destination (usually implementations limit a message to a single NoC packet), and for which software directly provides the source data to the NI (as values and not in memory).

Several variants of producer-consumer communication can result, depending on a number of factors:

- *Initiator* – producer-initiated (push type), or consumer-initiated (pull type) transfers.
- *Time of transfer* – when data are produced, when required by the consumer, or scheduled at some time between the two.
- *Location where data are placed* – locally to the producer, locally to the consumer, or at some "central" location.
- *Addresses of source and destination data* – same or different address used for the data copies created.
- *Data transfer granularity* – single word, cache-line, or larger blocks.
- *Synchronization method and transfer frequency* – separate signal by the producer, or integrated with the transfer. The second method may allow independent operation of the producer, without the need to wait for the transfer to complete.

Figure 2.2 shows some common communication patterns created by alternative choices for the above factors. Time is shown horizontally and space vertically. Synchronization is abstractly depicted with a green dashed line. Part (a) of the figure shows consumer-initiated (pull type) communication, corresponding to the case of invalidation-based coherent loads and stores. Part (b) of the figure shows the use of

[5]With write-send, special stores initiated by the producer are sent to the cache of a specific destination processor through the directory. Different instructions are provided in order to keep or invalidate the producer's local copy, and in order to write to main memory instead of to the cache of a destination processor. The directory invalidates other sharers or an exclusive owner, if required, and sends the data to the requested destination. The sending processor can utilize a write-coalescing buffer to aggregate writes to the same block before forwarding them.

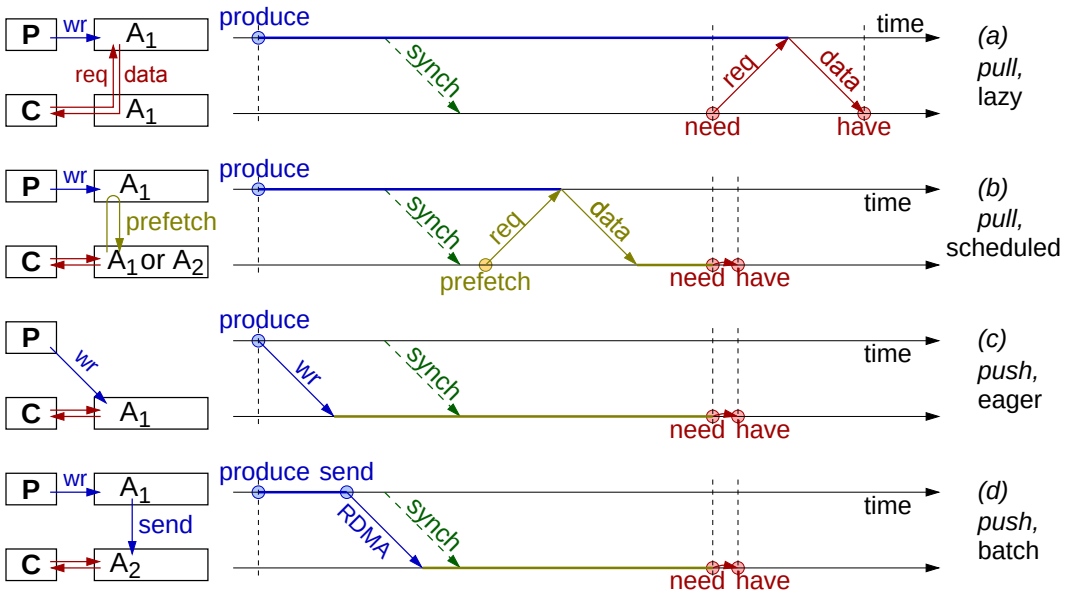


Figure 2.2: Communication patterns in space and time.

consumer-initiated prefetching or RDMA-read transfers. For cache-based prefetching, the copy has the same address (A_1) as the original data, while for RDMA-read the copy created has a different address (A_2). In both cases, the transfer is initiated before the data are needed, and is overlapped with other computation at the consumer. For such scheduled communication to more effectively hide latency, multiple pipelined transfers must be outstanding.

Part (c) of figure 2.2 shows producer-initiated (push-type) remote write transfers. Such transfers are provided by remote scratchpad stores, messages, write-send operations, and normal stores with write-update coherence. With the exception of messages, these mechanisms introduce the inefficiency of single-word transfers when used without a write-combining buffer. Even then, the application should provide an ordering of writes "convenient" for the write-combining bufffer. For write-update coherence, this kind of transfer is only possible when the consumer already had a copy of the cache-line at the time of the producer's stores. Remote writes economize memory utilization since they do not require any memory at the producer side, with the exception of messages that require a small buffer to construct the message (replacing a write buffer). Messages, though, can provide transfer atomicity and thus allow the inclusion of a synchronization flag with the transfer.

Finally, part (d) of the same figure shows producer-initiated, batch transfer of multiple words, supported by RDMA-write. This mechanism introduces the over-

head of first writing the data in local scratchpad memory, but can remedy inefficiencies caused by "inconvenient" write address order for a write-combining buffer, or lack of such a buffer. RDMA-write creates copies at addresses A_2 different from the source addresses A_1, and is efficient when a significant amount of the transferred data has been modified. Part (d) is the producer-initiated transfer corresponding to consumer-initiated prefetch of part (b) of the figure. In general, producer initiated transfers, as shown in parts (c) and (d), can occur immediately at or after production time. Synchronization takes place after the data transfer, opposite to the case of consumer-initiated communication.

2.2.1 Consumer-initiated Mechanisms

Figure 2.3 provides a more detailed illustration of consumer-initiated transfers. Time is again shown horizontally and space vertically. Except for the producer (P) and the consumer (C) timelines, a timeline has been added for the directory (D) in the case of coherent transfers. Labels write, (write/read) sync, (pre-)fetch, and read are provided over the producer's timeline for readability, denoting intervals for data production, synchronization, prefetch or RDMA-read, and data consumption respectively.

In part (a) of figure 2.3, which corresponds to figure 2.2(a), a pattern of loads and stores to three cache-lines over invalidation based coherence, is shown in close inspection. A non-blocking cache is assumed and weak ordering of events [44]. The producer fetches the lines, in order to update them, from a consumer, where they where transferred in a previous consume phase, thus stretching write time. In addition, before synchronization, the producer has to wait for its writes to complete, by issuing a fence, as illustrated with a thick red vertical line. Write time is also stretched because the fetch of updated lines has to go through the directory, increasing the latency of the last fetch which represents the communication overhead in write time.

Assuming the consumer was polling on the synchronization variable, a round-trip is required for the synchronization variable update. The illustrated scenario assumes optimistically, that the re-acquisition of the line with the synchronization variable by the consumer is overlapped with the producer's update request for the most part. This results in three traversals of the network between the producer to the consumer, at least one of which must go through the directory. After that, the consumer can read the lines, with an initial latency because of the round-trip for the consumer-initiated transfer and directory indirection, but this latency is overlapped

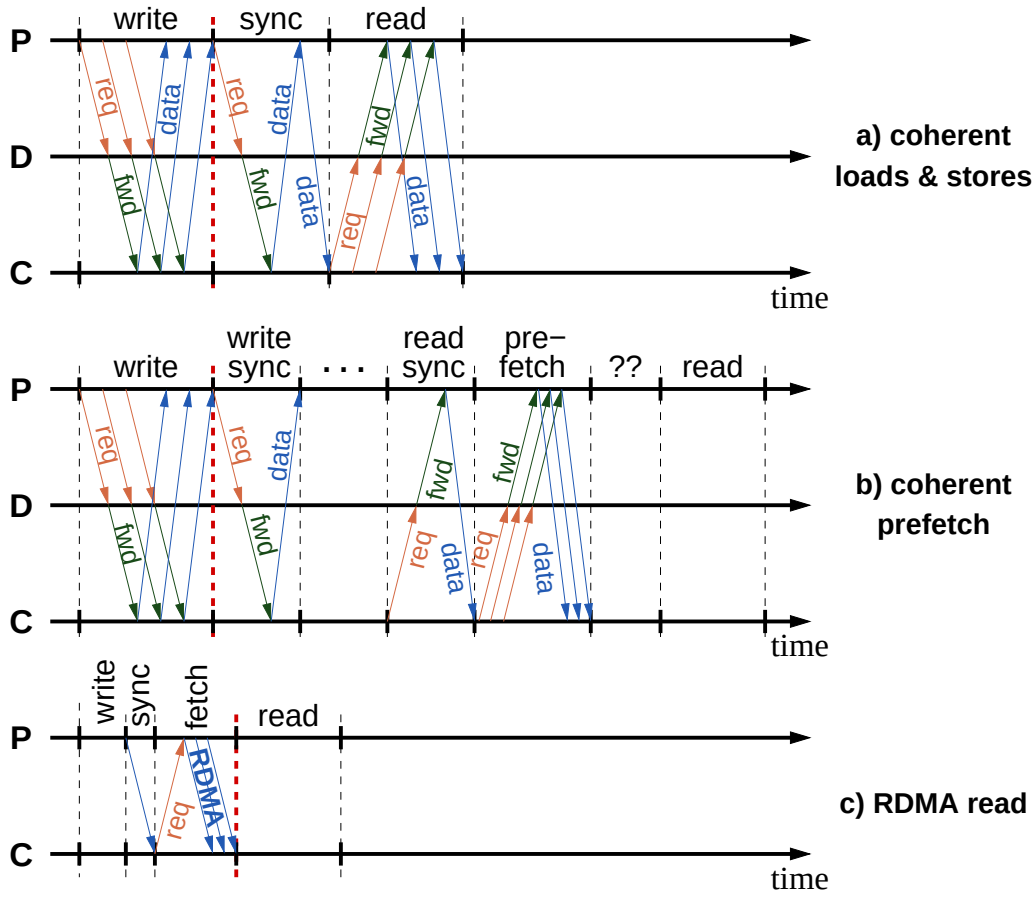


Figure 2.3: Consumer-initiated communication patterns in closer examination.

with the fetch of subsequent lines that can then be accessed with much smaller latencies.

Part (b) of figure 2.3, which corresponds to figure 2.2(b), shows closely a pattern utilizing coherent prefetching. The scenario presented refers to a programmable prefetcher, and not automated hardware prefetching. Write time is the same as in case (a), but the consumer will usually initiate the prefetch sufficient time after the producer completes, and thus synchronization will be broken in two parts, as illustrated: one after the data are produced, and one before the consumer initiates the prefetch.

The prefetch is a batch of overlapped requests (provided adequate non-blocking capability is supported in both the cache and the directory), followed by a batch of responses. There is no way for software to determine when the prefetch

has finished (a fence would normally exclude outstanding prefetched transfers), so how fast the read phase will be depends on how early the prefetch could be started. In the case of automated hardware prefetching, the prefetch and read time would be overlapped, increasing prefetching efficiency. In addition, hardware prefetching can overlap read time with back-to-back transfer of non-consecutive blocks.

Similarly, in part (c) of figure 2.3 a pattern for an RDMA-read transfer, that also corresponds to figure 2.2(b), is shown. In this case, write time is shorter because the write is local, there is no wait time to complete the writes, and a remote store can be used to signal the consumer, writing the synchronization variable with a direct transfer that minimizes synchronization time. The consumer does experience the round-trip for the consumer initiated transfer, but there is no directory indirection as for prefetching, and a batch of responses follows. With an ordered network, the consumer can poll on the last word of the specific transfer to detect completion, or a more complex mechanism is needed to implement a fence or transfer completion status update at the consumer. After data fetch is complete, read time is significantly reduced since the consumer works locally.

RDMA-read can be faster than coherent loads and stores with larger transfer sizes and with shorter completion times. Detecting RDMA completion with an unordered network, or with arbitrary size RDMA-copy operations, can be done using a hardware counter and partial transfer acknowledgements, as will be discussed in subsection 2.4.3. Similarly, completion notification latency will depend on whether the counter is located at the consumer or at the producer, the former being better since it only requires local acknowledgements for arriving DMA packets.

Overall, among consumer-initiated mechanisms, coherence-based implicit transfers utilize the same addresses at both the producer and the consumer, and because invalidation-based coherent accesses bring a copy locally, they will usually require to transfer data twice. Explicit RDMA-read employs two addresses, but because synchronization is at the unit of the total transferred data, it prevents automatic overlapping of communication with computation on the fetched data. Such overlapping has to be scheduled explicitly with computations on previous or subsequent data, using double or triple buffering. Thus, RDMA-read may be better suited for coarse-grain transfers. Conversely, because RDMA is an explicit transfer and its completion can be known to software, RDMA-read can be scheduled more effectively than prefetching. An advantage unique to automated hardware prefetching, is the batch transfer of non-consecutive blocks of memory, that is also overlapped with consumer's computation.

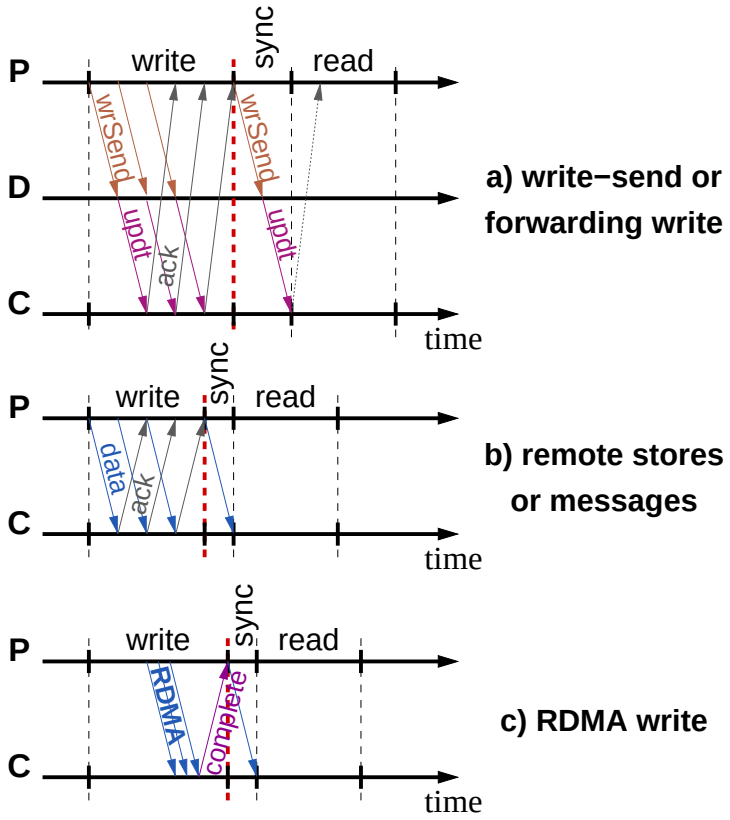


Figure 2.4: Producer-initiated communication patterns in closer examination.

2.2.2 Producer-initiated Mechanisms

Figure 2.4 illustrates in detail the producer-initiated transfers for the mechanisms under consideration. Part (a) shows the transfer of three cache lines with write-send, which corresponds to part (c) of figure 2.2. Similarly to case (a) in figure 2.3, the producer writing remotely the produced data experiences an increased write latency waiting for its writes to complete before synchronization. Again the writes have to go through the directory, which corresponds to an increase of the last write's latency that reflects the communication overhead in write time. Synchronization requires one trip to the consumer, also with directory indirection. Read time is reduced to the time for computation on local data.

Part (b) of the same figure shows in detail producer-initiated remote stores or messages, also corresponding to figure 2.2(c). Although in this case there is no directory indirection, the producer still has to wait for the acknowledgements of his writes before synchronization. Synchronization can be achieved at low latency

using a direct remote store. After that, read time is minimized by accessing local data. Alternatively, using messages that each includes a validity flag would reduce write time and obviate transfer overhead for synchronization. Although this may require additional read time for polling, such read time may be fully or partially overlapped with write time.

Part (c) of figure 2.4 shows producer-initiated RDMA-write corresponding to figure 2.2(d). Write time includes fast local write, the RDMA transfer, and waiting for its completion. After that, little synchronization time is required using a remote store. The read is minimized, as for all producer-initiated transfers, accessing local data. In subsections 2.4 and 2.4.1 we propose an optimization of this process, that removes both wait and synchronization overheads of an RDMA-write: the RDMA completion and the synchronization signal are combined with the use of a counter, and optimized by software placing the counter at the consumer. As with RDMA-read, RDMA-write efficiency improves with larger amounts of transferred data. For small, fine-grain transfers, messages or remote stores may provide the most efficient approach.

Overall, among producer-initiated explicit mechanisms, coherent write-send does not need to move data twice like normal coherent loads and stores. On the contrary, like other remote write mechanisms (see figure 2.2(c)), it economizes on memory by avoiding producer-side buffering. Despite that, directory indirection is required with write-send, which increases both the wait time of the producer before synchronization, and synchronization time itself, when compared to messages and remote-stores.

The use of RDMA-write prevents overlap of communication and computation on the transferred data, requiring synchronization for the complete transfer. For fine-grain transfers, RDMA-write is competitive to write-send, depending on initiation and synchronization overheads, and will outperform write-send, as the distance among communicating parties and the directory increases in many-cores processors. Given a large enough data transfer, RDMA-write will outperform write-send, messages, and possibly remote stores, because it employs direct and efficiently pipelined communication and amortizes initiation and synchronization overheads over the whole transfer. Reducing the time required for RDMA completion notification (as with counters) can also make RDMA competitive to remote stores even for fine-grain transfers.

2.2.3 Crosscuting Issues

Short message transfers provide the ability to optimize transfer completion by including a synchronization flag in the message. A similar optimization of the required synchronization has been proposed with coherent remote writes (similar to write-send and use write combining buffer at the sender) in [45]. In addition to supporting this optimization, messages preserve the advantage of being direct transfers, compared to write-send, and thus can be more efficient for fine-grain communication with increases in the distance among communicating parties and the directory. There are two potential disadvantages in using messages: (i) there is an overhead in constructing them, to provide the appropriate arguments to the NI, and (ii) their maximum size is limited. The latter is also true for cache line size.

Although figures 2.3 and 2.4 do not show all the packets required for coherent transfers, or all the cases of such transfers, it is apparent that coherence requires significantly more packets than explicit mechanisms supporting direct transfers, like the ones advocated here. This is true even if we consider acknowledgements for each RDMA packet, as recommended in subsection 2.4.2. For example, inspection of parts (b) and (c) of figure 2.4 to add the acknowledgements not present in (c), results in about the same number of packets for both, which is 2/3 of the number of packets required for write-send in part (a) of the same figure. A collaborative study of the author and others that also provides qualitative results [46], calculates that the number of on-chip packets used for coherent transfers is two to five times larger than the number of packets required for direct transfers.

2.3 Network Interface Design Alternatives for Explicit Communication in CMPs

Scalable on-chip networks define a new environment for network interfaces (NI) and communication of the interconnected devices. On-chip communicating nodes are usually involved in a computation, whose efficiency critically depends on the organization of resources as well as the communication architecture performance. Off-chip network architectures, in the LAN/WAN or PC cluster worlds, have been influenced by different factors and have evolved in different directions, with complex protocols, and robustness in mind, while latency was not a primary concern. From the perspective of the network interface, the on-chip setting has a lot more in common with supercomputers and multiprocessors targeting parallel computations.

Though systems-on-chip (SoC) utilize scalable networks, the device organization, NoC interface functionality and associated design tradeoffs are different than for chip multiprocessors (CMPs). For example, in SoCs, on-chip memory and NI resources are usually shared by many of the specialized computation engines, depending on requirements dictated by the targeted application or domain. In contrast, each processor core of a CMP usually has private access to on-chip SRAM for instructions and data, and some form of network interface directly serves the processor and these memories.

This chapter focuses on NoC interfaces for chip multiprocessors with a scalable on-chip interconnect, which represent an evolution of past supercomputer and multiprocessor off-chip network interfaces; it discusses how NI-CPU proximity allows optimizations in communication and synchronization, that improve computation efficiency.

The purpose served by the communication architecture is the efficiency of parallel computations. Parallel processing is based on interprocessor communication (IPC), which can be implicit or explicit, as discussed in section 2.1. Although implicit communication, supported by cache coherence is easier for the programmer, it may not allow for performance optimizations in those cases where the programmer, compiler, or runtime system, can manage locality better than coherence hardware. Explicit communication mechanisms are implemented in network interfaces, and enable the optimization of those cases.

This section follows the traditional notion that the network interface functions are those associated with explicit communication. Although the interface of *any* device to a network is literally and actually a "network interface", since this section refers to the design of network interfaces for explicit communication in CMPs, a more contextualized terminology is adopted. The term "network interface" is used only for the unit that supports explicit communication, while the more general term "NoC interface" is employed to refer to any kind of interface to the NoC, regardless if it supports explicit or implicit transfers. Thus, a network interface (i.e. one that supports explicit communication mechanisms) includes a NoC interface, just as any device connected to the NoC, but will also provide advanced functions, that are software programmable (or configurable), in addition to simple packetization and unpacketization of data, done in hardware.

Caches have a NoC interface, and have been optimized for the on-chip environment long before the multicore challenge became the focus of processor chip architecture. Although the hardware cost of caches is relatively small, and the processor's view of a consistent memory system based on caches has been studied extensively, the implicit communication style supported by cache coherence may not

scale well in terms of latency and energy efficiency, and the hardware cost of coherence directories is not negligible, as discussed in subsections 1.2.2 and 1.2.3.

Two main alternatives have been considered in the literature and implemented in CMPs for the network interface, differentiated by the naming scheme used to identify communication targets. The first is to directly name the destination *node* of communication, possibly also identifying one of a few queues or registers of a specific processor, but *not* any memory address. This results in a network interface tightly coupled to the processor, and may provide advantageous end-to-end communication latency, but usually necessitates receiver software processing for every message arrival.

The second approach is to place NI communication memory inside the processor's normal address space. In this case, communication destinations are identified via memory addresses, which include node ID information, and the routing mechanism may be augmented or combined with address translation. The resulting NI has the flexibility of larger send and receive on-chip space, that is directly accessible from software and allows the use of more advanced communication mechanisms. Challenges raised in the design of such a network interface include the possible interactions with the address translation mechanism, as well as managing the costs of full virtualization of NI control registers and on-chip memory, placed in a user-level context, including potential context swaps.

Both types of NoC interfaces will be described, with an emphasis on the second type which can support additional communication and synchronization mechanisms than simple messaging, and presents a more challenging design target.

2.3.1 Processor-Integrated Network Interfaces

The on-chip network can be directly interfaced to the processor pipeline by mapping it to processor registers (see figure 2.5). With this type of network interface, destinations can only be determined by their node ID[6], or via register or queue numbers that includes a node ID, and no naming or addressing mechanism is provided to identify memory of a remote processor or network interface. Any NI buffers are *private* to the sending and receiving nodes, and are managed by hardware. Restricted local access to such buffers may be provided, through processor privileged operations, to support context switches.

Only a send-receive style of communication is supported with this type of network interface. Processors send and receive messages directly from a subset of their

[6]The virtualized variant of this approach, will use a thread ID instead of a node ID.

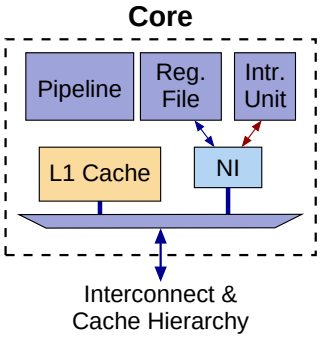


Figure 2.5: NI directly interfaced to processor registers.

registers, which essentially act as NI registers. Hence, communication operands appear directly in these registers, while the compiler cannot use them for general purposes. The processor is occupied during message send proportionally to the transfer size, and may *block* because of NoC or receiver occupancy. Messages are usually of a small, fixed maximum size.

Receive-side processing is normally required for every communication event (i.e. message). The processor can block on reading from an *empty* register, or, user-level reception interrupts may be used. Alternatively, an asynchronous privileged interrupt can be used to remove incoming data from the NI, and allow subsequent processing by software. This mechanism has the overhead of copying incoming traffic, and is typically only used for overflow handling. Furthermore, privileged reception interrupt cost in out-of-order processors can be very high because of outstanding memory operations, unless a special mechanism that squashes non-committed instructions is provided for asynchronous interrupts[7].

Multi-word messages are usually atomic in the sense that individual words of the message are presented to the receiver as a unified entity, and thus message transfer appears as a single transaction. For the initiation, an explicit message send instruction is usually provided, that posts the whole message towards its destination once it is completed. This type of interface minimizes transfer latency, because communication is directly initiated from and arrives to *private* processor registers, avoiding processor interaction with any intermediate device. Bandwidth is limited to at most one processor word per cycle and potentially reduced by transfer initiation overheads.

In past multi-chip multi-processor systems, iWarp [47] interfaced the network

[7]Modern processors provide mechanisms to enter the kernel that do not change the context and have much smaller overhead. These mechanisms, though, are processor synchronous.

directly to processor registers in support of *systolic communication* and in the MIT J-Machine [48] similar mechanisms were used for fast remote handler dispatch. More recently, this approach has been used in CMPs, in the MIT Multi-ALU Processor (MAP) [49] to exploit fine-grain parallelism and provide concurrent event handling via multithreading, and in Tilera's TILE64 chip [7] in support of operand networks. In the Imagine stream processor [50] the network for inter-cluster communication is mapped to the stream register file. Processor-interfaced *asynchronous direct messaging (ADM)* has also been proposed, for fast task synchronization support in CMPs [51].

In order to provide decoupling of the sending and receiving processors with this type of NI, some buffer space is usually provided at receivers. To keep the buffering requirements low and avoid dropping messages, the NI must provide a mechanism for *end-to-end flow control*, that will prevent or retry message transmission when no space is available at the destination. In case protected processor context switches must be supported, the newly scheduled thread should not have access to the buffer contents for the previously scheduled thread. To provide a clean receive buffer on context switches, either supervisor software must have access to the buffer contents, or a hardware mechanism should support copying of received messages to off-chip memory.

In addition, the NI must guarantee the delivery to software of the packets stored in its receive buffers, so that transactions complete and NI buffers are freed. When this is not guaranteed, a form of protocol deadlock specific to message passing systems can take place. If two hardware threads continuously send messages to each other, without removing arriving messages from the network interface, receive buffers will eventually become full. Network buffers will then fill up because of backpressure, and the threads will, eventually, deadlock when no more messages can be injected into the network. To prevent erroneous or malicious software from this kind of deadlock, when the NI receive buffer is almost full, some mechanism must guarantee its contents are copied to off-chip storage. This can be done, again, either by an automatic hardware facility, or by a high priority system thread that is scheduled-in via a receive buffer overflow interrupt [52]. It is also possible to allow user-level interrupt handlers to process incoming messages, but such handlers must be restricted from actions that may cause them to block, e.g. by acquiring locks or sending messages [53].

Regardless of the mechanism that guarantees software delivery of messages, buggy or malicious sender software could be filling receive buffers faster than the receiver could do any useful processing of incoming traffic. Once receive buffers become full, backpressure will incrementally clog the network with packets, inter-

fering with the performance of other threads. There are two possible approaches to this problem: automatic off-chip buffering and end-to-end flow control. Providing an automated facility to copy messages off-chip, may require costly rate-matching on-chip buffers to inhibit affecting other threads, and requires an independent or high priority channel for the transfer to off-chip storage.

Alternatively, the NI may provide some form of end-to-end flow control that can keep the buffering requirements low. Providing some guaranteed receive side buffering per sender and acknowledged transfers for end-to-end flow control, would be viable only in small systems. One possible solution is provided by limited buffering at the receiver for all senders, and sender buffering for retransmission on a negative acknowledgment (NACK). Such sender-side buffers should also be flushed on a context switch, if message ordering is required. Hardware acknowledgments (ACKs and NACKs) for end-to-end flow control require a network channel independent to the one used for messages, to avoid deadlock.

Because software usually requires point-to-point ordering of messages, if the NoC does not provide ordered delivery of packets (e.g. because of using an adaptive or other multipath routing scheme), the NI must provide reordering at receivers. Keeping the cost of reordering hardware low in this case, probably requires that the amount of in-flight messages per sender is kept very low. Even if the network supports in-order packet delivery, the NI may still need to provide some support for point-to-point ordering when the end-to-end flow control mechanism involves NACKs and retransmissions [51].

2.3.2 Network Interface Integration at Top Memory Hierarchy Levels

Explicit inter-processor communication can also be supported with a network interface integrated at one of the higher levels in the memory hierarchy, close to the processor. This is usually some kind of scratchpad memory –also called a local store in the literature. Parts (a) and (b) of figure 2.6 show placement at L1 and private L2 cache levels, for memory hierarchy integration of the NI. Part (c) of the same figure, illustrates a cache-integrated NI at the level of a private L2 cache, using portions of cache memory for scratchpad and exploiting the common functionality.

In cases (b) and (c), the processor TLB (not shown) can be used to distinguish cacheable and scratchpad accesses, and, possibly, to prevent the L1 data cache form responding to scratchpad accesses. In these cases, corresponding to L2-cache level integration, local scratchpad memory may or may not be cacheable in the L1. If L1

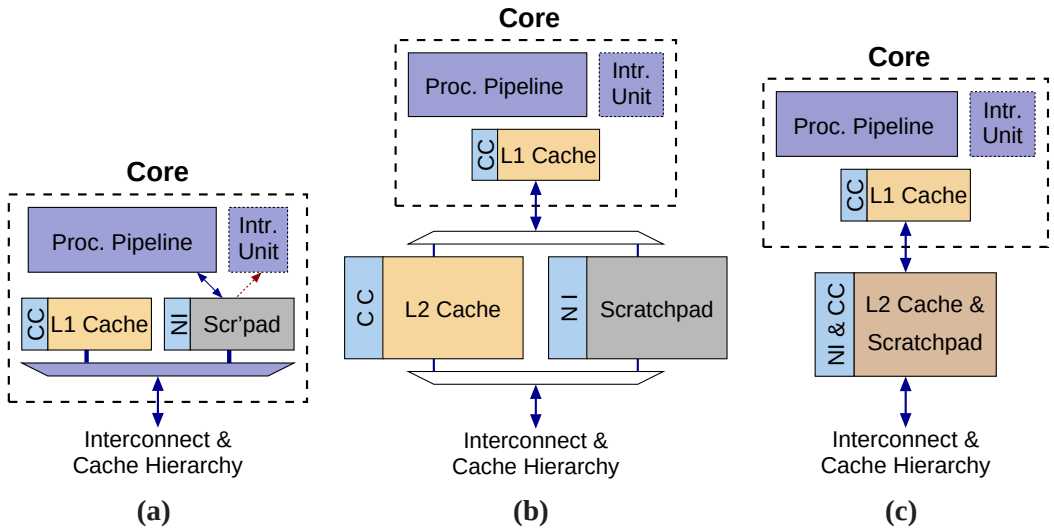


Figure 2.6: Microarchitecture of NI integrated at top memory hierarchy levels.

caching is allowed for local scratchpad memory, the L1 may need to be invalidated on remote scratchpad accesses.

A less aggressive placement of the network interface, further from the processor, at some level of the memory hierarchy shared by a group of cores, is also perceivable. In this case, processor access of the NI can be uncached (as in most network interfaces for off-chip communication), or may exploit coherence mechanisms [35]. In this case, the NI would mostly be utilized for explicit transfers among parts of the hierarchy belonging to different processor groups. This organization, though, is not studied here.

With a network interface integrated at top memory hierarchy levels, communication operations always have their source, destination, or both in memory. Scratchpad memories provide sufficient buffering for bulk transfers, and thus the NI may support RDMA (get and put) or copy operations. It is natural to provide access to the local scratchpad with processor loads and stores. When the processor supports global addresses (i.e. processor addresses are wide enough), or if remote scratchpad memory can be mapped in its address space, direct load/store access to remote scratchpads can also be provided. In any case, atomic multi-word messages can be supported as well.

At least one of the source and destination operands of communication operations always resides in memory shared by processors other than the one initiating a transfer. For this reason, synchronization is required, both before the transfer and for its completion, with the processor(s) that may be accessing the shared operands

concurrently. Conversely, because application data can be placed in NI memory, communication can occur asynchronously with computation (i.e. without occupying the processor), both for transmission and for reception. The aforementioned communication mechanisms, with the exception of scratchpad loads and stores, can provide such decoupling of NI communication operations and processor computation, effectively allowing software to overlap multiple transfers with each other and with computation.

The latency of short transfers is generally increased compared to processor-integrated NIs, because transmitted operands may need to be written to the local scratchpad before departure, and received operands must be accessed from scratchpad after synchronization, as opposed to sending and receiving from processor registers directly. On the contrary, as long as scratchpad memory and network resources are not busy, NI bulk transfers can exploit the full NoC bandwidth, which may oftentimes be two or four processor words per cycle[8].

Moreover, it is possible for this type of NI to also support blocking scratchpad loads to minimize reception overhead and reception interrupt mechanisms to allow arbitrary processing to occur asynchronously (shown only in figure 2.6(a) with a red arrow), as provided with SPE channels in the CellBE [13]. Normally, though, polling must be used to detect operand reception for communication operations other than scratchpad loads.

In multi-chip multiprocessors this type of NI was exploited for the sender-side of the AP1000 [54] to transfer cache lines, and there are a few recent CMP designs that also take this approach. These include the CellBE [13], from IBM, Sony, and Toshiba, implementing eight synergistic processing elements (SPE) with private scratchpad memories inside a global address space accessible via coherent RDMA; Intel's Single-chip Cloud Computing (SCC) experimental 48-core chip [5] which provides 8 KB/core on-chip message passing buffers (MPB), accessible by all cores via loads and stores, as a conceptual shared buffer inside the system address space; and the cache-integrated NI for the SARC European IP project [55] presented in this thesis, implemented in a multicore FPGA-based prototype [56, 57], which supports all the communication mechanisms that will be discussed in subsection 2.3.4 and a set of synchronization mechanisms presented in section 2.4.

NIs integrated at top memory hierarchy levels provide explicit communication and synchronization, but are related to caches in that NI communication memory is "visible" to the application. With this kind of NI, instead of separate processor

[8]This results in reduced latency for bulk transfers that exceed some minimum size, provided that scratchpad memory bandwidth is equal or exceeds that of the NoC.

and NI memory resources, on-chip memory is shared and better utilized, obviating the redundant copy of communication operands that was common in traditional off-chip network interfaces with dedicated memory. With memory hierarchy integration there are no dedicated buffering resources managed exclusively in hardware. As a result, producer-consumer decoupling and flow control, supported in hardware with processor-integrated NIs, may need to be arranged under software control. For example, single and multiple reader queues supported in the SARC network interface are implemented in memory shared by the NI and the processor, and are managed by hardware and software in coordination –i.e. the NI only supports ACKs and flow control is left to the application.

Load and store accesses to local or remote scratchpad memory and to NI control registers must utilize the processor TLB for protection and to identify their actual target. The destination of these operations can be determined from the physical address or by enhancing the TLB with explicit locality information –i.e. with extra bits identifying local and remote scratchpad regions. In the case of a scratchpad positioned at the L1-cache level (figure 2.6(a)), TLB access may necessitate deferred execution of local scratchpad stores for "tag" matching, as is usual for L1 cache store-hits.

Although scratchpad memory can be virtualized in the same way as any other memory region, utilizing access control via the TLB, the system may need to migrate its contents more often to better utilize on-chip resources. For example, scheduling a new context on a processor may need to move application data from scratchpad to off-chip storage, in order to free on-chip space for the newly scheduled thread. Migration of application scratchpad data is a complicated and potentially slow process, because any ongoing transfers destined to this application memory need to be handled somehow, for the migration process to proceed. Furthermore, mappings of this memory in TLBs throughout the system need to be invalidated. To facilitate TLB invalidation, the NI may provide a mechanism that allows the completion of in-progress transfers without initiating new ones.

In a parallel computation, consumers need to know the availability of input data in order to initiate processing that uses them. Reversely, producers may need to know when data are consumed in order to replenish them, essentially managing flow control in software. For these purposes the NI should provide a mechanism for producers and consumers to determine transfer completion. Such a mechanism is also required –and becomes more complicated– if the on-chip network does not support point-to-point ordering. Write transfers that require only a single NoC packet can be handled with simple acknowledgments, but multi-packet RDMA is more challenging. Furthermore, generalized copy operations, if supported, may be initiated by one

NI and performed by another –e.g. node A may initiate a copy from the scratchpad of node B to the scratchpad of node C. Chapter 2 in subsection 2.4.3 shows how all these cases can be handled using acknowledgements and hardware counters.

Since NI communication memory coincides with application memory, software delivery of write transfer operands is implied, as long as NoC packets reach their destinations. Nevertheless, the NI needs to guarantee that read requests can always be delivered to the node that will source the response data without risking network deadlock. Networks guarantee deadlock-free operation as long as destinations sink arriving packets. The end-nodes should be able to eventually remove packets from the network, regardless if backpressure prevents the injection of their own packets. Because nodes sinking read requests need to send one or more write packets in reply, reads "tie" together the incoming and the outgoing network, effectively not sinking the read, which may lead to what is called protocol deadlock.

To remedy this situation the NI may use for responses (writes in this case) a network channel (virtual or physical) whose progress is *independent* to that of the channel for requests (reads). As long as responses are always sunk by NIs, the request channel will eventually make progress without deadlock. Alternatively, reads may use the same network channel with writes, but they need to be buffered at the node that will then source the data. The reply of the read will then be posted by that node, similar to a locally initiated write transfer, when the network channel is available. Finally, for transfer completion detection, the NI may need to generate acknowledgments for all write packets. These acknowledgments must also use an independent network channel and NIs must always be able to sink such packets.

Since bulk transfers are offloaded to the NI while the processor can continue computation and memory access, a weak memory consistency model is implied. When posting a bulk transfer to the NI, subsequent memory accesses by the processor must be considered as concurrent to NI operation (i.e. no ordering can be assumed for NI and processor operations) until completion information for the bulk transfer is conveyed to the processor. To provide such synchronization, the NI must support *fences*[9] or other mechanisms, to inform software of individual transfer completion or the completion of transfer groups.

When remote scratchpad loads and stores are supported, these accesses may need to comply with a memory consistency model [58]. For example, for *sequential*

[9] As discussed in chapter 2, in the context of the processor, fences or *memory barriers* are instructions that postpone the initiation of subsequent (in program order) memory operations, until previous ones have been acknowledged by the memory system and thus are completed. Similarly, the network interface can provide operations that expose to software the completion of previously issued transfers.

consistency [59] it may be necessary to only issue a load or store (remote or local) after all previous processor accesses have completed, sacrificing performance. With a more relaxed consistency model like weak ordering of events [44], fences are required and special synchronizing accesses. In addition, the NI may need to implicitly support ordering for load and store accesses to the same address, so that the processor expected order of operations is preserved and read-after-write, write-after-read, or write-after-write hazards are avoided.

Additional complexities arise because the network interface resides outside of the processor environment, in the memory system. The network interface may need to tolerate potential reordering of load and store operations to NI control registers by the compiler or an out-of-order processor. For example, the NI should be able to handle a situation where the explicit initiation of a message send operation and the operands of the message arrive in an unexpected order. Finally, out-of-order processors may issue multiple remote scratchpad loads, which may require NI buffering of the outstanding operations, in a structure similar to cache miss status holding registers (MSHR).

A final thing to note is that in the presence of cacheable memory regions, all NI communication mechanisms may need to interact with coherence directories, which, in turn, will need to support a number of protocol extensions. To avoid complicating directory processing, it may be preferable that explicit transfers to coherent memory regions are segmented at destination offsets that are multiples of cache-line size and are aligned according to their destination address (i.e. use destination alignment explained later in this chapter).

2.3.3 NI Control Registers and Virtualization

Virtualization of the network interface allows the OS to hide the physical device from a user thread, while providing controlled access to a virtual one. The virtualized device must be accessible in a protected manner, to prevent threads that belong to different protection domains from interfering with each other. Protection is managed by the OS, and can allow control on how sharing of the physical device is enforced. For higher performance, the virtual device should provide direct access to the physical one, without requiring OS intervention in the common case, which is referred to as *user-level access*, and is necessary for the utility of NI communication and synchronization mechanisms in the on-chip environment of CMPs. NIs accessible at user-level allow parallel access to the physical device by multiple threads, without the need for synchronization.

The mechanisms that provide such synchronization-free sharing of the physical device and delegate control of the sharers to the operating system, are implemented in hardware. For example, a mechanism commonly used to provide user-level access is memory mapping of device resources. Access control, provided via address translation and protection hardware, allows the required OS control over the memory-mapped resources. Virtualization usually requires additional hardware support, to allow OS handling of thread context switches. The OS must preserve the state of the switched out thread, including any communication state, so that the thread can later resume transparently. It must also guarantee that, after the context switch is completed, the physical device allows unobstructed use by other threads.

Traditionally, NI control registers are used to provide multi-word or block transfer descriptions to the NI, which cannot be programmed with a single instruction. Such descriptions may refer to messages and RDMA or copy operations. In addition, special NI control registers can support synchronization, providing atomic operations, blocking processor access, receive-side queuing and configuration of communication event interrupts.

The control registers of a processor-integrated NI, are a subset of the processor registers. Because send operations are processor synchronous, only a single set of such registers is adequate for the supported transfers of short messages and scalar operands. Virtualization of the NI requires that any send and receive buffers associated with these control registers are brought to a consistent state and freed when another thread is scheduled on the processor, and can later be restored to that state when the initial thread is rescheduled. In the case of multithreaded processors, NI registers and associated buffers need to be replicated per hardware thread.

In a network interface integrated at the top memory hierarchy levels, control and status registers provide a processor-asynchronous interface, and can thus be more versatile, especially when several communication and synchronization mechanisms are supported. Moreover, communication overlap benefits from multiple sets of control registers, that can allow multiple outstanding transfers. For virtualization, a number of copies of the control and status register sets is needed. Memory mapping of an NI register set in the address space of a process, allows low overhead, user-level access. The number of register sets defines the number of processes that can access the NI concurrently. In case more processes need simultaneous access to the NI, the OS will need to resort to frequent context switches to timeshare the use of the register sets. In addition, the number of control register sets per thread, limits the number of supported concurrent transfers. The cost of virtualization is proportional to the number of control register copies.

An alternative is to provide configurability of scratchpad memory, so as to

allow programmable control "registers" inside the scratchpad address space. This is feasible by providing a few tag bits per *scratchpad line* (block). The tag bits are used to designate the varying memory access semantics required for the different *line types* (i.e. control "registers" and normal scratchpad memory). The resulting memory organization is similar to that of a cache and can support virtualization via address translation and protection for accesses to the scratchpad memory range.

This design allows a large number of control "registers" to be allocated inside the (virtual) address space of a process. The NI can keep track of outstanding operations by means of a linked list of communication control "registers", formed inside scratchpad memory. Alternatively, the total number of scratchpad lines configured as communication "registers" may be restricted to the number of outstanding jobs that can be handled by a fixed storage NI *job list*. Such a job list processes in FIFO order transfer descriptions provided in NI control "registers" –potentially recycling in progress RDMA transfers, segmented in multiple packets. Scaling the size of such a job list structure, and thus the number of supported outstanding transfers, results in low hardware complexity increase, in contrast to outstanding transfers for a cache that require a transaction buffer or miss status holding register (MSHR) fully associative structure.

Virtualization of explicit inter-processor communication, also requires handling of virtual destinations. In the case of processor-integrated NIs, a translation mechanism similar to address translation, i.e. a hardware *thread translation table* in the NI, can map threads to processors, filled by the OS on misses. In addition, the NI must be aware of the thread currently scheduled on the processor, for which it can accept messages, and a mechanism is required to handle messages destined to threads other than the scheduled one. For example, Sanchez et al. [51] have proposed the use of NACKs and *lazy invalidation* of the corresponding entry in the thread translation table of the source processor.

The case of NIs integrated at top levels of the cache hierarchy that support RDMA or copy operations, requires that virtual address arguments can be passed to NI control registers, although in a protected way that is compatible with page migration. A few solutions have been proposed to the NI address translation problem [60, 61, 62], but processor proximity of the NI may simplify the situation. A TLB (or MMU) structure can be implemented in the NI to support the required functions. Updates of address mappings may use memory mapped operations by a potentially remote processor handling NI translation misses, (e.g. in the Cell BE SPE translation misses are handled by the PPE). Alternatively, access to a second port of the local processor's TLB may be provided.

Furthermore, transfers destined to a node that arrive after the node's migration

need to be prevented; the usual approach to this is a TLB sootdown. To reduce the cost of TLB invalidations generally, SARC architecture proposes progressive address translation, as discussed in chapter 3 and in [63]. Finally, to support migration of scratchpad regions that include lines marked as NI "registers", the operating system may read and record tag bits of the region at migration time. Alternatively, lines marked as NI "registers" may be recorded by the OS at the time of their allocation. To optimize the migration process, the OS may restrict NI "register" allocation inside special scratchpad pages.

A crucial question, regarding both the execution and the implementation overhead of related mechanisms, is how often will thread[10] migration be, or, said otherwise, *why migrate a thread in a many-core processor?* One possible reason is to switch-in another thread. Threads of the same process should probably be co-scheduled to cooperate, unless there is dramatic load imbalance among them; thus a context switch is not likely to involve threads of the same process, and even if this occurs, there is the alternative of mapping in the same cache scratchpad resources of more than one threads. A context switch with a thread of another process could be avoided if there are other available cores. If the load is excessive, one can also consider sharing the same node without migrating resources.

There is also the case a thread is idle for a long period, which should be infrequent and, unless heavily required, need not swap out scratchpad contents necessarily. Finally, a situation that may require migration of a thread is when part of the chip becomes "hot". This should be quite seldom, as there are other mechanisms to lower the activity in a part of the chip (e.g. voltage/frequency scaling), but in this case the penalty of migration can be tolerated. This is because temperature raises slowly, and migration may be on the order of microseconds (Cell SPEs take about 20 microseconds for a context switch).

2.3.4 Explicit Communication Mechanisms

Explicit communication mechanisms supported by CMP network interfaces can provide direct point-to-point transfers and thus offer efficient communication and minimize energy per transfer. Bulk explicit transfers can be used for macroscopic software prefetching and to overlap multiple transfers and computation, while messages and direct loads and stores to scratchpad can provide low latency signaling and common data access. The two network interface types presented have largely different

[10] Main-memory page migration should be seldom, especially if page contents are interleaved at fine-grain in multiple DRAMs (see figure 1.2). Scratchpad page data will be handled at user-level, but scratchpad storage (or part of it) will be usually associated with a specific thread.

properties. Processor integrated NIs provide very low latency interactions, based on send-receive style transfers that are interfaced directly to the processor. NI integration at top memory hierarchy levels provides adequate space for more advanced communication functions, and allows the read-write communication style as well.

The former NI type provides register-to-register transfers and dedicated, hardware managed resources, while the latter supports memory-based communication and shares NI resources and their management with the application. As will be described below, in the first case, the possibility of protocol deadlock involves the receiving processor and interrupts, or potentially expensive support for automatic off-chip buffering. In the second case, read requests may require resources for an independent subnetwork, or receiver buffering resources for a limited number of read requests, that can be delegated to the application. Both NI types require an independent subnetwork for acknowledgements to provide flow control and other functions.

Messaging-like mechanisms can be used for scalar operand exchange, atomic multi-word control information transfers, or, combined with a user-level interrupt mechanism at the receiver, for remote handler invocation. In processor-integrated NIs, message transmission directly uses register values. An explicit *send* instruction or an instruction that identifies setting of the final register operand is used to initiate the transfer. At the receiver the message is delivered directly to processor registers.

In the case of a network interface integrated at the top memory hierarchy levels, the message must be posted to NI registers and the transfer can be initiated either explicitly via an additional control register access or implicitly by NI monitoring of the transfer size and the number of posted operands. At the receiver the message can be delivered in scratchpad memory or in NI control registers used for synchronization purposes.

Loads and stores to local or remote scratchpad regions can be started after TLB access, depending on the consistency model. Remote stores can use write-combining to economize on NoC bandwidth and energy. In this case, the processor interface to the NI would include an additional path from the combining buffer (not shown in figure 2.6). Although it is possible to acknowledge stores to scratchpad to the processor immediately after TLB access, scratchpad accesses must adhere to the program dependences, as discussed in the previous section. Remote loads can be treated as read DMA requests, or exploit an independent network channel for their single-packet reply, avoiding the possibility of protocol-level deadlock. In support of vector accelerators, the NI may also provide multi-word scratchpad accesses.

When the NI resides below the L1 cache level (see figure 2.6(b) and (c)),

scratchpad memory may be L1 cacheable. Coherence of scratchpad and cached copies may be kept by hardware or software. For example, the SARC cache-integrated network interface presented in this dissertation, resides in a private L2 cache, as in the case of figure 2.6(c). Scratchpad memory "locked" in the L2 can be cached only in the local L1, which is write through and is invalidated appropriately on remote writes to scratchpad. In the case of the Single-chip Cloud Computing (SCC) Intel experimental chip, per core message passing buffers (scratchpad memories) can be cached without coherence in L1s throughout the chip. The L1s support a single cycle operation that allows software to rapidly invalidate all scratchpad copies locally cached.

The remote direct memory access (RDMA) mechanism implements *get* and *put* bulk transfers to and from the local scratchpad. The direction of the transfer (read or write) is explicit in RDMA, which requires that software is aware of both local and global addresses. Specifying an RDMA transfer initiated by a remote NI is not fully supported in the usual case. For example, the SPE DMA engine of the Cell processor utilizes *local store* and *effective* addresses and corresponding commands, while for transfers initiated by the Power (or peripheral) processor element (PPE) each SPE has a separate DMA queue where RDMA operations can be placed remotely.

Write DMA commands that include the source and destination addresses of the transfer, as well as its size and opcode, need to be buffered in appropriate NI control registers to keep track of operation progress. A large write DMA transfer should be segmented and processed in multiple iterations. This will avoid blocking other processor traffic (messages or remote stores) while the DMA is in progress, and will allow interleaving segments of multiple outstanding DMAs for parallel progress of transfers toward potentially different destinations.

For read DMA the receiving NI must be able to remove the incoming request from the NoC and generate the appropriate reply. In this case, not blocking the input channel while sending a multi-packet response can be accomplished by breaking the read request in multiple read packets so that each requires a "short" reply. Alternatively, the NI can buffer read DMA requests until they are fully processed, which obviates the need for separate read and write network channels, but limits the number of concurrent reads the NI can support depending on the amount of buffering provided.

With this second approach, if the available buffer space is exceeded, the NI needs to *drop* superfluous requests, record the event and notify the nearby processor of the error condition (e.g. with an interrupt). The SARC network interface employs a software-provided buffer in scratchpad memory for read requests, delegating to the software the responsibility to provide a buffer large enough for its needs. A

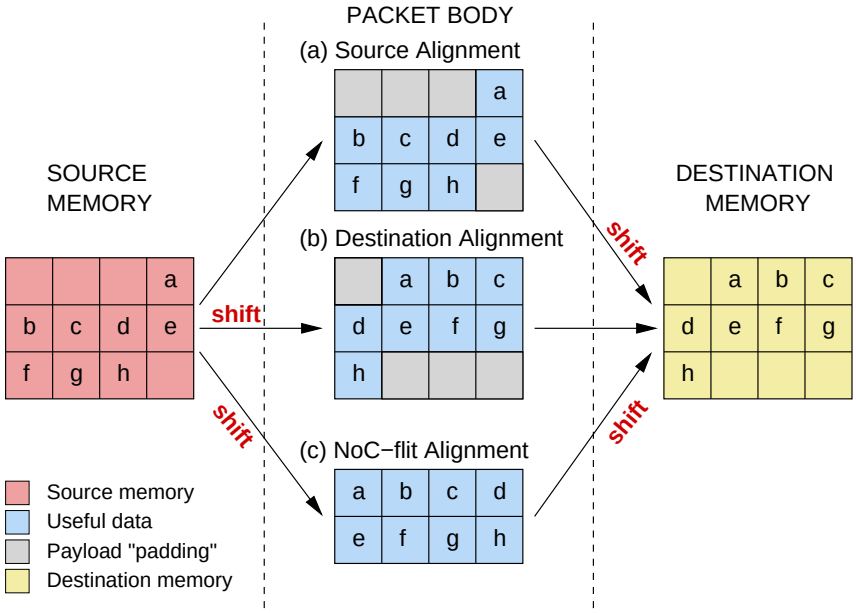


Figure 2.7: Three alternatives of handling transfers with arbitrary alignment change.

complementary approach for a read request exceeding the available buffer space at the NI hosting the read's source region, would be to employ some kind of negative acknowledgement that manifests the error condition at the initiating node.

When scratchpads can only be accessed via a global address space, it is more natural not to limit the sources and destinations of communication mechanisms to the local scratchpad. This precludes get and put operations (RDMA), and the congruent mechanism provided by the NI supports a *copy* operation. Copy operations are more general than RDMA because they allow the specification of transfers that have both remote source and destination addresses. Notification of the initiating node for transfer completion in hardware is more complex in this case.

RDMA and copy operations that allow arbitrary changes of data alignment may be dealt with in three different ways, as illustrated in figure 2.7. First, it is possible to send write data in network packets keeping their *source alignment*, as shown in Figure 2.7(a). This implies that packets may have "padding" both at the beginning (before useful data) and at the end. In addition, a barrel shifter is required at the receiver to provide the operation requested alignment. Second, if we choose to send packets with their requested *destination alignment*, as shown in Figure 2.7(b), then "padding" may also be required both at the beginning and at the end, and a barrel shifter must be placed at the source node.

Third, we may want to minimize the number of NoC flits transferred, as shown

in figure 2.7(c), in which case we need two barrel shifters: one at the source node to align the transmitted data to the NoC flit size boundary, and another barrel shifter at the destination node to fix the requested destination alignment. In this case, packets can only have "padding" after useful data. This third approach is more expensive, requiring two barrel shifters per node, and cannot reduce the amount of transferred data by more than a single NoC flit. Restricting the supported alignment granularity reduces the cost of the barrel shifter circuit, but complicates software use of RDMA or copy operations.

Transfer pipelining is important for both latency and bandwidth efficiency of transfers. For instance, NoC request scheduling can overlap with NoC injection of a previous packet. To keep the latency of communication mechanisms to a minimum, it is important that the network interface implements cut-through for both outgoing and incoming packets. When the maximum packet size does not exceed the width of the receiver's memory, cut-through implementation for the incoming path can speculatively advance the tail pointer of the NoC interfacing FIFO for packet reception, until the correct CRC is computed at the end. Alternatively, for NIs integrated at the top of the memory hierarchy, providing separate CRC for the packet's header and body allows writing incoming packets if the destination address is correct before checking if the data CRC is correct.

2.3.5 Producer-Consumer Synchronization and Transfer Ordering Support

When using explicit transfers, software needs to actively manage the ordering of operations. Because of this, explicit communication can become crabbed in the case of processor-asynchronous handling of bulk transfers by the NI, or when the on-chip network does not support ordering. To simplify the handling of operation ordering the NI should support both efficient and straightforward transfer completion detection. In addition, in the context of producer-consumer communication, detection of data arrival by the consumer should be both flexible and fast to allow efficient fine-grain interactions. This special case of transfer completion detection by the consumer is addressed as *data synchronization* in the following.

Detection of data reception by a consumer (i.e. data synchronization) is necessary before a computation on communicated data can start. For this purpose NI designers usually optimize data synchronization combining it with data reception. This is true for send-receive style communication and consumer-initiated transfers. Reversely, in the case of producer-initiated transfers, individual transfer completion

information enables the initiating node to synchronize with the consumer as required, and enforce point-to-point ordering when it is not supported by the NoC or the NIs. The common element of these mechanisms, that enables producer-consumer interactions and allows exploitation of NI-initiated transfers, is transfer completion detection. Providing this type of information, without compromising NI scalability and communication performance, may be a challenging goal for network interface design.

There are three basic mechanisms for application software to detect message reception:

1. *Blocking receive operations.* This first mechanism is based on NI support for blocking read access to NI registers, until a message arrives. A blocking "receive" operations can provide the lowest reception latency when the operation is issued before message arrival. Software must express correctly an order of issuing send and receive operations that does not cause deadlock (e.g. two nodes exchanging values should not both do a blocking "receive" before sending across their values). Blocking operations can be combined with processor transition to a low power state for energy efficiency.
2. *Polling on NI registers or scratchpad memory.* The second mechanism avoids blocking by means of "peek" operations on NI status registers or scratchpad memory, where a message or an acknowledgment is expected. This provides non-blocking reception handlers and allows computation to proceed while waiting for a message. Polling should be local to reduce reception overhead and to avoid congesting the network. The potential downside of this mechanism is that for purposes other than event processing, the appropriate frequency of polling is very difficult to assess effectively and may introduce unnecessary software overhead.
3. *Interrupting the destination processor.* The third mechanism invokes a user-level interrupt that forces the execution of a message reception handler. Dispatch of the appropriate handler can be automated with hardwired information in special message types [48]. User-level interrupts have lower overhead than privileged ones, but event handling with interrupts is not straightforward. The user needs to disable interrupts when synchronous processing atomicity must be provided [34]. In addition, the user-level interrupt mechanism may require that the handler removes the received message from NI dedicated storage without blocking (e.g. by sending messages or acquiring locks) [53]. Nevertheless, this last restriction can be relaxed by providing a higher priority privileged interrupt mechanism that handles NI resource overflow and

underflow transparently [51].

Processor-integrated NIs usually support point-to-point ordering transparently, in order to support operand passing semantics, reduce on-chip communication overhead, and be programmed more naturally. Data synchronization is provided by the message reception mechanisms discussed above. NIs integrated at top memory hierarchy levels can –for the most part– provide the same data synchronization mechanisms for messaging, exploiting load and store accesses to scratchpad or to NI "registers".

Considering read-write instead of send-receive communication style, similar mechanisms can be provided for RDMA operations through NI control registers, both on the producer and the consumer side. These mechanisms though, may require additional support to detect transfer completion, whose complexity can vary depending on a number of factors: (i) the number of outstanding accesses supported, (ii) RDMA semantics, and (iii) whether or not the NoC supports point-to-point ordering. A large number of outstanding accesses requires a lot of state (and a lengthy transfer ID) to track their progress and completion. With copy semantics, more than two nodes may be involved in the transfer and its completion, since a shared address space is implied. Similarly, arbitrary size virtual DMA semantics may also involve source or destination regions that reside in multiple nodes. Lastly, with an unordered network, point-to-point completion signals need to be grouped and counted appropriately.

In addition, with the read-write communication style (i.e. for load and store as well as for RDMA-copy operations), the memory consistency model interferes with synchronization. At best, a weak consistency model must be assumed since bulk multi-packet transfers would be difficult or inefficient to handle otherwise, especially with an arbitrary size. In this case, the NI can provide fence operations that guarantee the completion of all previously initiated transfers. Separate fence operations can be supported for each mechanism to provide more flexibility. For example, a simple fence mechanism for scratchpad stores only requires a counter at the initiating NI that keeps track of the arithmetic sum of departures minus acknowledgments.

Individual operation completion can also be exposed to software, providing *explicit transfer acknowledgements*. This can be supported by allowing software to specify an acknowledgement address for each communication operation, at which an acknowledgment value will be deposited upon transfer completion. This mechanism is sufficient to establish message ordering over an unordered network, but multi-packet RDMA or copy operations require additional support to confirm that

all associated packets have been delivered to their destination. The counter synchronization primitive is introduced for this purpose in section 2.4.1, and can provide the flexibility of selective fences that identify the transfer of software task operands.

For producer-consumer interactions, synchronization declares the event of new (produced) data to the consumer. This event "intervenes" between the producer's write or send and the consumer's read or receive. Some transfer mechanisms can combine this event with the data transfer, and in such cases synchronization time is minimized. Such mechanisms are:

(i) register-to-register (send-receive style) transfers where the consumer (receiver) is blocked, or receives a user-level interrupt;

(ii) memory-based messaging where the consumer is polling on a flag included in the message;

(iii) RDMA over an ordered network that similarly uses a flag included in the transfer and consumer polling;

(iv) RDMA-copy transfers utilizing the proposed counter synchronization primitives, where the consumer polls on a flag that automatically accepts a notification from the counter.

It is also possible to allow polling in case (i), based on implementations such as that of TILE64 chip [7], or as proposed by Sanchez et al. [51]. Tilera's chip could support polling, based on automatic off-chip buffering of incoming traffic, but the actual instruction set is not disclosed. The proposal of Sanchez et al. supports a peek instruction, returning the message source and length of an incoming message, based on privileged overflow interrupts and end-to-end flow control in hardware.

Similarly, mechanisms that support blocking or interrupts in cases (ii), (iii), and (iv), are also feasible, as with SPE channels in the Cell processor for case (iii). Such mechanisms replace the simple flag referenced in cases (ii), (iii), and (iv), with NI control registers associated with the transfer. These mechanisms, though, are not a good match for bulk RDMA transfers; blocking better suits fine-grain transfers and interrupts fit well with infrequent processing.

The mechanism of case (i) above, covers synchronization for producer-consumer interactions with register-to-register transfers. NIs integrated at top memory hierarchy levels support the read-write communication style and corresponding mechanisms. In the case of producer-initiated communication, synchronization requires notifying the consumer about transfer completion. For all mechanisms, the data are transferred to scratchpad "near" the consumer. Remote stores require that

the producer is first notified of transfer completion and then writes a flag at the consumer. Cases (ii), (iii), and (iv) above, correspond to producer-initiated messages, and RDMA or copy operations. If the flag optimization is not used and transfer completion is detected only at the producer, he must write a separate flag at the consumer, after the transfer is complete.

Reversely, in the case of consumer-initiated communication, the producer must first write the data locally, and then write a flag at the consumer. This allows the latter to initiate the transfer, either with remote loads or with RDMA (read or copy). Consumer-initiated transfers require synchronization that is as expensive as in the cases of producer-initiated transfers that cannot exploit the combined or automatic flag update optimizations in (ii), (iii), and (iv) above.

The network interface can support synchronization mechanisms that *multiplex* data from multiple senders (producers) in an NI queue at the receiver (consumer), when ordering among producers is not important. This allows the consumer to avoid searching for arriving data in multiple places. In addition, multiple such queues can be provided, to allow a receiving node to *demultiplex* messages, according to some categorization. For example, messages from the same sender, or messages carrying a stream of data, or messages of some specific type, can be automatically categorized by hardware in order to be processed in a uniform or orchestrated manner. The first property of such synchronization support (multiplexing of senders) is natural in processor-integrated NIs, which queue messages from multiple senders. Its second property (message demultiplexing in categories) is natural in memory hierarchy integrated NIs, which demultiplex arriving packets to different addresses.

Tilera's TILE64 multicore chip [7] supports a small number of queues in a processor integrated NI. Demultiplexing is enabled via hardware supported message tags specified by the sending node. Queues at the receiver allow software settable tags for tag matching in hardware, and an additional queue is provided for unmatched messages. The SARC prototype (section 3.2) provides similar functionality, allowing an arbitrary number of *single-reader queues* to be allocated in scratchpad memory, and supporting multiple queue item granularities (see subsection 2.4.1). Each queue operates as a single point of reception, multiplexing arriving messages, in support of many-to-one communication. A discusion of how single-reader queues can optimize buffering requirements proportionally to the usual (or expected) number of possible senders, instead of the maximum, is provided in subsection 2.4.3. While SARC network interface requires software handling of flow control and the receiver reads data from scratchpad, Tilera's processor-integrated NI provides flow control in hardware and data are found in registers.

2.3.6 Synchronization and Consistency in a Shared Address Space

The consideration of producer-consumer communication alone, as handled in the previous section, can hide a global aspect of synchronization with memory-based transfers. Producer-initiated transfers in figure 2.2 *seem* to avoid the read/prefetch overhead of consumer-initiated ones. The actual situation is quite more involved.

In many cases, a consumer receives data from many producers at about the same time, and a producer distributing data to many consumers is also common. In addition, it is usual producers and consumers to exchange roles during the execution of a computation, and most of the time consumers are simultaneously producers. This means that ***coordination*** is required among producers and consumers to manage the memory of the consumers, and that it can be usual for all processors to be involved in multiple such coordinations at about the same time.

With consumer-initiated transfers, the consumer does this management, naturally, at the time data are requested, and potentially notifies the producer(s) to free their copy after the transfer completes. With producer-initiated transfers, which can be more efficient, there is a complication: the consumer may not be available at the time one or more producers are ready to provide data.

NIs for off-chip communication provide a range of approaches to this issue. Such NIs have been oriented to explicit communication, and may or may not utilize shared addresses. In the cases that do, that is in support of RDMA and a sparse shared address space is used only for explicit transfers and is managed as a pool of "pinned" and registered pages and not as an address space. Handling buffers at the consumer is done through memory registration and, in Infiniband, with queue pairs (e.g. [64]) and posting receive requests to the NI [65]. Overall, with this approach, handling of consumer memory is based on previous negotiation of a producer and a consumer.

One approach that employs shared addresses and solves the problem of consumer buffer space, has been proposed in the context of indirect explicit communication and data-forwarding mechanisms, such as write-send discussed in section 2.2. For example, in [40], write-send operations arriving at a remote cache can cause the write-back of another cache-line. Depending on the memory hierarchy architecture, this may or may not require additional independent (sub)networks than those required for the basic coherence protocol without write-send.

A hybrid approach that combines direct and indirect communication has been proposed for direct cache access [66]. Although direct cache access has been pro-

posed to optimize the performance of I/O intensive workloads, by selectively pushing into a processor's cache IP packet header DMA data, it can also be considered for user-level data, explicitly transferred between private parts of a cache hierarchy. Variants that allocate or only update cache lines in the target cache are evaluated in [66], and also an alternative version of the proposed technique is discussed, where only prefetch hints are posted to or snooped by the target cache.

NIs for off-chip communication that do not use a shared address space backed by memory, which provide mostly a send-receive two-sided model, often employ a large *matching space* to independently buffer traffic categories in queues (e.g. [67]). Their mechanisms have similarities to tagged queues in Tilera's TILE64 chip, although the tag space in processor-integrated NIs of TILE64 chip is very small. Single-reader queues with software flow control provide an alternative implementation, but require copying data out of the queue, that also fits mostly a send-receive communication style. Handling consumer buffers, especially for unexpected transfers (sends arriving before the corresponding receive is posted), and avoiding copies additional to the one from NI to processor memory, is one of the main reasons past NIs for off-chip communication utilize large dedicated memories. These approaches do not fit very well to the on-chip limited space and the efficient use of a shared address space.

An elegant and flexible approach was proposed in [52], where a user-level interrupt mechanism at the receiver is combined with automated buffering in virtual memory. Although this requires an independent path or (sub)network to main memory, it is possibly the approach taken in Tilera's TILE64 chip. Coupling of the NI in the cache hierarchy, with a coherent network interface [35], can also support NI buffer overflow to main memory, transparently to software and potentially in a virtualized way. This approach mostly fits NIs loosely-coupled with respect to the processor.

Nevertheless, adopting either direct NI access, as in [52], or via cache coherence mechanisms, as in [35], for on-chip interprocessor communication in manycore processors, may introduce interference issues in a multiprocess environment, that may be less pronounced with off-chip communication and had not been studied in the past. These issues have been addressed in [51] with NACK-based flow-control and an interrupt-based NI for on-chip interprocessor communication. These two approaches, that can solve the problem of consumer on-chip space management, are for NIs that manage dedicated private buffers or queues, and match well with a send-receive communication style.

Overall, and to the best of the author's knowledge, there are four possible approaches to consumer memory management for producer-initiated transfers and NIs

managing and exploiting an on-chip shared address space:

(1) *Statically or in advance arrange consumer buffer space for all producers.* Occasionally, at the beginning of a computation, by providing sufficient consumer memory for every possible producer requirement. At times, some form of flow control can be exploited as well, requiring per producer buffers. The most rewarding approach is to organize computation in phases, so that buffers corresponding to specific producers during one phase, have a different assignment in another phase, and this assignment is explicit in the algorithm. Unfortunately, this is not always possible, as well.

(2) *Query the consumer, before the transfer, with a message that generates a user-level interrupt (i.e. revert back to a consumer-initiated transfer, with an extra network crossing).* The consumer can try to find some available buffer to service the producer's request. In case there is no space available, and it is too costly or not possible, to move data and free space, the consumer can delegate resolution of the situation to the future, sending a negative response. This negative response constitutes a form of backpressure to the producer.

(3) *Handle a consumer's memory as a resource equally shared among all threads, and have all interested parties coordinate dynamically regarding its use.* This is the usual approach taken by shared memory algorithms, treating all memory, including a consumer's memory, as "*equally*" shared. One reason for this is that the shared address space creates the impression of equidistant memory. As a result algorithms tend to describe independent producer and consumer operation, with on-demand search for the opportunity to access the equally shared resources. This on-demand search for opportunity requires a global interaction of the interested parties to coordinate such accesses. The usual means of coordination are locks. Depending on the structure of the shared resources, coordination can be fine-grain, or even wait-free.

(4) *Employ a mechanism for automated spilling of on-chip NI buffers or memory to off-chip main memory.* This approach is elegant and can solve the problem transparently, in hardware. It may require, though, an independent path or (sub)network to main memory, and even then, rate matching buffers should be provided in that path to avoid hotspots or interference with other traffic.

The first approach is efficient, roughly, as long as the working set of producers and consumers can fit in scratchpads. Beyond that point, a mechanism is required to take action dynamically, and spill data at other hierarchy levels or to off-chip memory. This problem underlines the need for locality management mechanisms in

the network interface.

The second approach is quite general and flexible, and indicates that it may be very useful to incorporate such interrupts associated with messaging. It is possible for the consumer accepting the interrupt, to initiate the transfer with RDMA-read to its local memory, acknowledging the producer. The consumer can even arrange the release of space with notification to the producer and complete request handling. Nevertheless, execution time overhead would match or exceed that for consumer-initiated transfers. Space would have to be allocated for messages from all possible producers, so a reasonable place to associate such a mechanism for interrupts would be a single-reader queue instead of the message itself, so that a consumer can provide the address of this special queue to desirable producers or senders.

The third approach is also very general and flexible. Coordination has two potential advantages: (i) it can allow a producer to acquire guaranteed access to multiple consumer buffers (or arbitrary memory locations) at once; (ii) it can tolerate a thread that is a consumer for multiple buffers (or arbitrary memory locations) that need not be local to any one place, preventing producer updates to them. In addition, because memory is equally shared by all processors (i.e. as if no physical proximity exists) a consumer has a limited degree of freedom on consuming data that are not strictly local (i.e. not in his local scratchpad). Actually, from a correctness standpoint, the data consumed can be arbitrarily far. It is easy, though, for this last property to encourage computations that can be programmed efficiently exploiting locality, to use algorithms that view memory uniformly and are usually inefficient.

The fourth approach has not yet been proposed for an explicitly-managed shared address space, but only in the contexts of private NI memory resource management, and of coherence-based data-forwarding, for an implicitly-managed shared address space. Management of hotspot and interference issues can become a problem since off-chip bandwidth cannot keep in pace with on-chip bandwidth with current technology. To avert hotspot or interference problems, negative acknowledgements (NACKs) can be used with end-to-end flow control, as in [51], but this requires outstanding transfer state at the sender and would limit sender injection rate and bandwidth for reasonable implementations.

Note that none of the above alternatives requires that consumers compute only on local data. In comparison of the second and third alternatives, using locks (or other global coordination mechanisms) in a *small* “neighborhood” of processors, to manage producing and consuming data within the “neighborhood”, may be more efficient than having processors in the “neighborhood” keep track of data location and send to each other messages that cause interrupts. It is probable, though, that this will not be always the case as the “neighborhood” size increases.

A subtle point to observe is the following: when using a shared address space, departure from a computation confined to a set of pairwise interactions that are bounded with synchronizations, results in the need for a model of global memory consistency. When only pairwise interactions and correlated synchronizations are enforced, synchronization and flow control suffice for memory consistency. Such a computation model would associate synchronizations with a "transfer of ownership" of the related data, implicit in the program. This requires only point-to-point ordering, and can be provided for writes with an ordered network, or with acknowledgements and counters over an unordered NoC. In the general case, it also requires that reads wait for all previous node writes (local and remote) and for synchronization.

The model of computation described, includes, but it is not limited to, the case of computing only on local data. When restricted to computation only on local data, remote loads are of very limited use if any, and RDMA-read and read messages must be associated with synchronization. In this restricted case, local loads need only wait for local writes or synchronization. This fits better with simple RDMA and the model of the Cell processor than to a shared address space with general RDMA-copy support and remote loads.

With the general shared memory model data races are allowed; that is multiple concurrent accesses to shared data, at least one of which is a write. This is in contrast to the case of computations limited to pairwise interactions that are bounded by synchronizations; the "memory model" for computations of the latter type does not allow any competing accesses (i.e. data races) other than for synchronization. Because of data races, a memory model that provides a form of *global* consistency is required with shared memory, in the general case, which can introduce additional overheads to producers, as seen in section 2.2. This has similarities with the case of producer-consumer interactions bounded by synchronizations over an unordered network, but in the latter case the only requirement for synchronization operations at transfer boundaries is transfer completion notification.

The equally shared memory approach of approach (3) above, does not prevent pairwise interactions, but it allows races which may introduce overheads to producer-initiated transfers. The consumer memory management problem is solved in this case by assuming "infinite" equidistant memory. More importantly, producer-consumer communication is difficult to identify and associate with synchronization, because of shared data structures. Such data structures must be read and written in a consistent fashion, that requires coordination of the many and potentially concurrent producers and consumers sharing them. The latter is crucial, because it involves two ordering problems: (i) the order of a producer's updates to shared state, and (ii) the

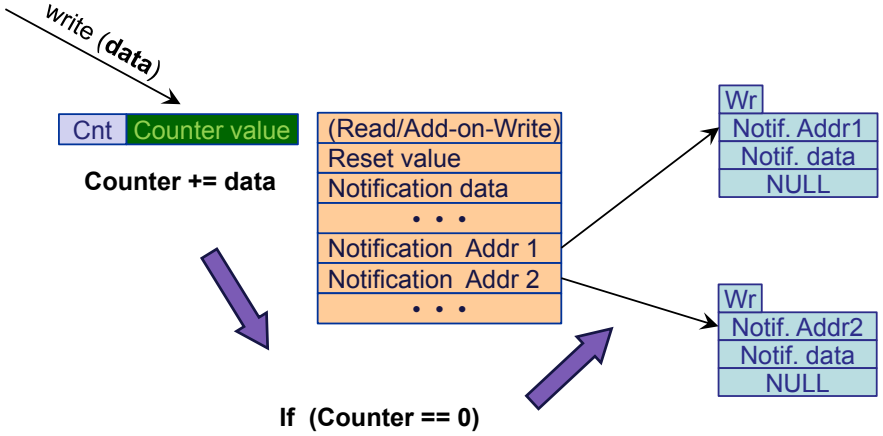


Figure 2.8: A counter synchronization primitive atomically adds stored values to the contents of the counter. If the counter becomes zero, a number of notifications are sent by the network interface. The actual counter value is stored in network interface metadata, and a small portion of NI memory is associated with each counter to provide a software interface for accessing the counter, and configuring its operation.

order these updates are observed by a consumer.

2.4 Support for Direct Synchronization

2.4.1 The Operation of Direct Synchronization Primitives

Two types of hardware primitives for synchronization are proposed, in support of explicit communication in a shared address space: counters and queues. For queues, two variants are introduced, one for a single reader and one for multiple readers. Counters are intended for the management of sequences of unordered event. Single-reader queues provide support for efficient of many-to-one communication, and novel multiple-reader queues are designed for the pairwise matching of producers with consumers, when speed is important and specific correlation is not.

Counters support the anticipation of a number of events that fulfill a condition. Reaching the expected number of events, triggers automated reset of the counter and notification of possible “actors” waiting the condition to be satisfied. Figure 2.8 shows the operation of a counter. The illustration assumes that the counter value is kept in network interface metadata, shown in light mauve and green, and a small block of NI memory is associated with the counter to provide an interface for soft-

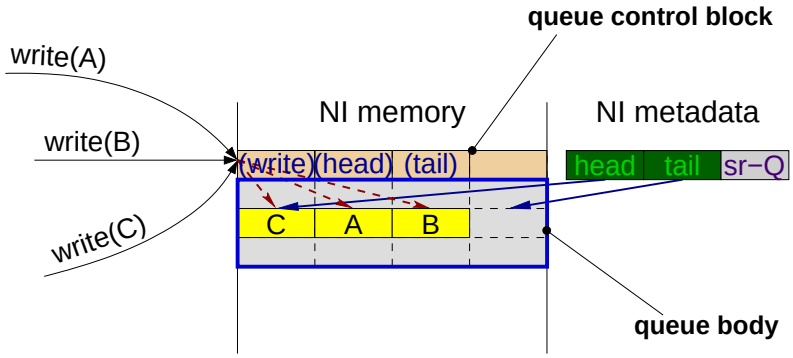


Figure 2.9: A single-reader queue (sr-Q) is multiplexing write messages, atomically advancing its tail pointer. Both the head and the tail of the queue are kept in NI metadata, while the queue body is formed in NI memory. The head pointer is managed by the local processor accessing it through a control block in NI memory.

ware manipulation of the counter, which is shown in light orange. Software uses different offsets inside that block, to configure notification data and addresses, and a reset value for the counter. One of the offsets provides indirect read access to the counter and ability to modify its value.

Counter modification is done by writing a value, and results in increment of the counter by that value. If after this modification the counter becomes zero, notifications are sent to all the addresses specified in the block associated with the counter. Notification packets, shown in light blue in the figure, write the notification data specified (a single word) to the notification addresses, and suppress returned acknowledgements using a NULL acknowledgement address[11].

Single-reader queues support the automated multiplexing of data from multiple senders in hardware, minimizing receiver effort to track down new data arrival. Data arriving to the single-reader queue (sr-Q) are written to the queue offset pointed by the queue tail pointer, atomically incrementing it, as depicted in figure 2.9. A control block is provided in NI memory as an interface to the queue, which is formed in adjacent NI memory. Different offsets of the control block allow writing to the queue, reading the tail pointer, as well as reading and writing the head pointer. The latter two are intended for the local processor. The actual head and tail pointers of the queue are kept in NI metadata.

To access the sr-Q the local processor keeps shadow copies of the head and tail pointers, reading data from where the former points to. To dequeue an item,

[11]We assume read and write type packets for explicit communication, that carry an *explicit* acknowledgment address.

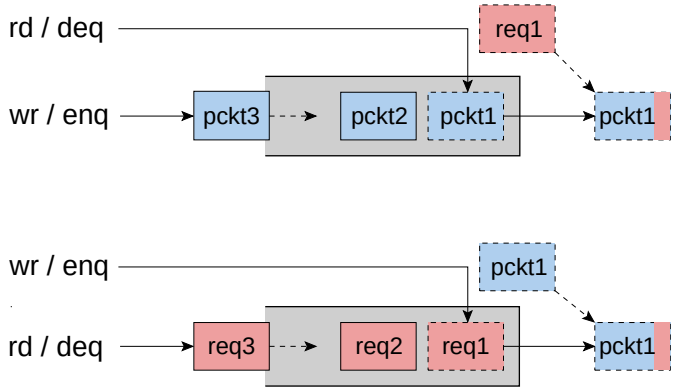


Figure 2.10: Conceptual operation of a multiple-reader queue buffering either read or write packets.

the processor advances the shadow head and stores it to the appropriate offset of the control block. When the shadow head pointer becomes equal to the shadow tail, the processor starts polling[12] the actual tail pointer until the latter advances. At that point new data have arrived, hence the processor updates its shadow tail and restarts processing queue data. For flow control, the single reader must signal each writer after his data are dequeued. The space required is proportional to the number of writers and multiple queue item granularities can be supported for access via messages.

Multiple-reader queues combine multiplexing of short message data from multiple producers and multiplexing of requests from multiple consumers, buffering either in the same queue. They signal the availability of both by serving a request with matched message data, even when the request comes first, acting as a *binary semaphore* [68] for exclusive acquisition of message data. Conceptually, the multiple-reader queue (mr-Q) buffers *either* data *or* requests, as illustrated in figure 2.10. When data are stored and a request arrives, shown in the upper part of the figure, the data on the top of the queue are sent to the response address for the request. When requests are buffered in the queue and data arrive, shown in the lower part of the figure, the data are forwarded to the response address for the top request in the queue. In either case, when a new item arrives (request or data), that is of the same type to those already buffered in the queue, it is also buffered at the tail of the

[12] Single-reader queues can also provide a mechanism that blocks the processor until the queue becomes non-empty. It is also desirable to provide a way for the blocked processor to transition to a low-power state, and possibly to associate such a processor state with more than a single queue that can become non-empty and cause the processor to recover its normal operation mode. Low-power states are not provided on FPGAs, and thus our prototype does not implement such mechanisms.

queue.

From the perspective of software, a multiple-reader queue can be viewed as a novel *FIFO with dequeuer buffering*. In addition, it can be accessed with non-blocking read and write messages, disengaging the processor. Because dequeue (read) requests constitute queue items exactly as enqueued (write) data, the mr-Q can amortize the synchronization that would be required to check if the queue is empty for a dequeue, using non-blocking messages. Viewed otherwise, the mr-Q never returns empty, but instead supports delayed dequeues, which are buffered until corresponding data are enqueued.

Processor access to the mr-Q's head and tail pointers (not shown in figure 2.10) is avoided, and can hardly be of any use since multiple concurrent enqueues and dequeues can be in progress. Enqueuers and dequeuers can utilize a locally managed window of enqueues and dequeues respectively, to guarantee that the queue never becomes full by either enqueues or dequeues. For dequeuers, this is done by assuring that the count of outstanding dequeues (the reads minus the responses) does not exceed their window (a fixed space granted in the queue). A dequeuer getting an enqueuer's data from the queue needs to notify the enqueuer of the space released in the queue, and the latter must assure that the count of enqueues minus the notifications he receives does not exceed his window (a corresponding queue space granted in the queue). The space required for the queue in this case, is proportional to the maximum of the readers and the writers. Other more complex flow control schemes may also be possible.

The approach described for flow control above, provides concurrent access to the mr-Q, exploiting direct transfers to allow synchronization among the contenders with only local management. An inefficiency of this scheme appears when dequeuers do not immediately inspect a dequeue's response so as to notify the enqueuer. This inefficiency though is amortized to the total size of an enqueuer's locally managed window of enqueues. Hardware support for negative acknowledgements, discussed in subsection 2.4.2, can enable single- and multiple-reader queue space management that is independent of the number of potential senders, as will be discussed in subsection 2.4.3.

2.4.2 Access Semantics and Explicit Acknowledgements

The synchronization primitives of the previous section and the associated control space for counters and single-reader queues, should be accessible in NI scratchpad memory, via a scratchpad part of the address space. Figure 2.11 illustrates

the resulting remote access semantics to different *types* of NI memory and how acknowledgements are generated for each NoC packet. Three types of NI memory are depicted in the middle, with read and write request packets arriving from the left, and the corresponding generated reply packets on the right. The different access semantics are combined with an extended message passing protocol, in which read packets generate write packets and write packets generate acknowledgements. Acknowledgements are designed to support completion notification for single- and multi-packet transfers. Because of this, read and write packets must allow an acknowledgement address which can be provided by user software.

As shown in the figure, a read packet arriving to normal scratchpad memory has the usual read semantics and generates a write reply packet carrying the destination and acknowledgement addresses in the read. In the case of an RDMA-copy operation generating the read packet, multiple write packets will be generated to consecutive destination segments. Read packets to counters also generate a write packet in reply –not shown. A read packet arriving at a multiple-reader queue has dequeue semantics and the write reply may be delayed.

Writes arriving at normal scratchpad memory have the usual write semantics, unlike writes to counters that have atomic increment semantics and writes arriving to queues that have enqueue semantics. Nevertheless, in all cases, a write packet generates an acknowledgement (ACK) toward the acknowledgement address in the write, carrying the size of the written data as the data of the acknowledgment. This size can be accumulated in counters for completion notification of the initial transfer request (read or write). This is because acknowledgements arriving at any type of NI memory act as writes (not shown), but do not generate further acknowledgements. Note that the acknowledgement generated for a write to a multiple-reader queue is sent immediately after data are written in the queue and *not* when they are dequeued[13].

Negative acknowledgements (NACKs) can be interesting for a number of purposes. Our prototype, described in section 3.2, does not implement NACKs for simplicity reasons, but we partially evaluate their use in simulations (see subsection 4.3.2). NACKs can be exploited in the case of a remote read to a single-reader queue, which is something we would like to forbid. They can also be used for writes to both queue types, as well as reads to multiple-reader queues, that cause queue overflow.

In addition, they can support the semantics of a special *unbuffered read*, which can be usefull in the case of multiple-reader queues –for other types of NI memory it

[13] Although both alternatives could be provided under software configuration of the mr-Q, we choose to preserve the same semantics for acknowledgements of writes to any type of NI memory.

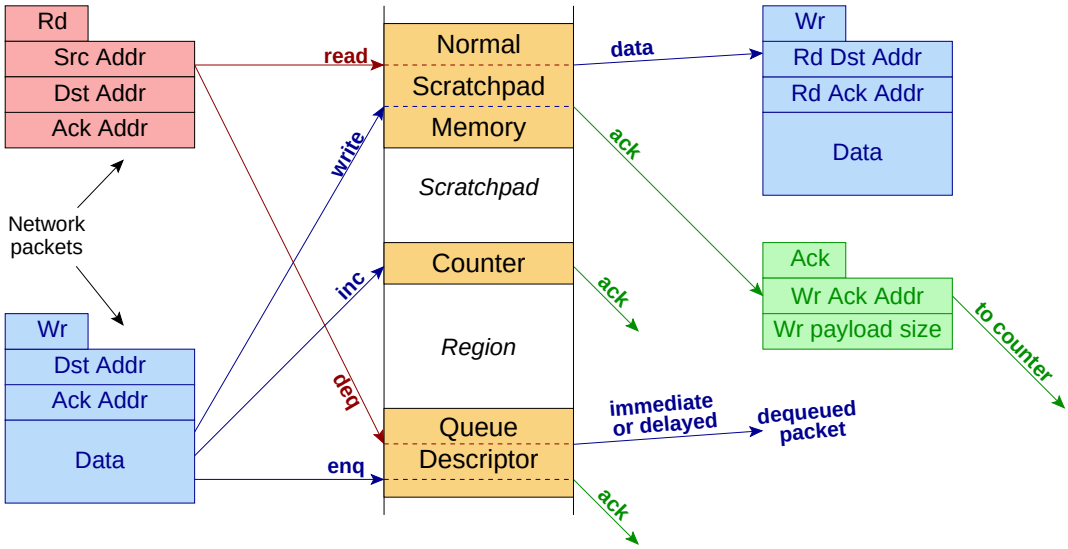


Figure 2.11: Remote access to scratchpad regions and generation of explicit acknowledgements.

would default to a normal read. An unbuffered read to an mr-Q would immediately reply with a special NACK if no data were pending in the queue. This enables a dequeue-try from the mr-Q that does not need to wait if no data are yet enqueued. Similarly, an *unbuffered write* that generates a special NACK if there is no dequeue request pending in the queue, can also be considered. In all cases, a NACK implies that the read or the write was *not* buffered in the mr-Q. Such unbuffered reads and writes to a multiple-reader queue will always return either ACK or NACK, regardless of the progress of other threads.

2.4.3 The Use of Direct Synchronization Mechanisms

Single-reader queues can be used to optimize many-to-one communication via multiplexing of senders, and forming multiple such queues in scratchpad memory allows demultiplexing of message categories, as discussed in subsection 2.3.5. Compared to RDMA, single-reader queues have two disadvantages, making them more suitable for control information exchange than actual data transfers: (i) they require copying of data out of the queue and into the program's data structures[14], and (ii) they fix data processing order to the order of arrival. Single-reader queues also have

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

an advantage compared to RDMA: they require constant time for locating arrived input. As discussed earlier (subsection 2.1.2), RDMA requires per sender buffering at the receiver, and thus locating arrived input requires polling time proportional to the number of possible senders.

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

Single-reader queues are very closely-related to producer-consumer communication, in any pattern (i.e. one-to-one, many-to-one, or many-to-many). Philosophically, one can view counters and multiple-reader queues as abstractions of different attributes of single-reader queues; counters abstract away the data buffered in a single-reader queue and only store “arrival events”, while multiple-reader queues abstract the reception order imposed by queue implementation, making them suitable for multiple concurrent readers. Next we discuss the relation of counters and multiple-reader queues to synchronization and some of their applications.

Synchronization between computing entities (i.e. threads) is in general required in two different situations: (i) when resolving data dependencies, and (ii) when modifying the state of shared resources using more than a single operation. In the first case, the dependent computation must wait for the write or read of its input or output arguments before it can proceed. This includes all types of dependencies: true data dependencies, anti-dependencies, and output dependencies. The synchronization in this case involves one or more *pairs* of computing entities.

The second case usually appears for complex shared data structures or devices (or complex use of such resources), and involves many (potentially all) computing entities sharing the resource. It occurs because the required processing consists of multiple, inseparable operations on the resource, which we cannot or do not want to separate. This is usually required to preserve the usage semantics of an interface or implement the definition of a complex operator.

For example, usually we would not want to separate the operations required to insert an element in a shared balanced tree and re-balance the tree. In this situation, we would need to preclude other operations from accessing some part of the tree. Precluding others from accessing the resource is needed to keep the intended operations inseparable (i.e. without intervention), and is usually addressed with the use

of locks. In general, what is required is consensus among contending sharers, on an ordering of their operations on the resource that does not violate the intended usage semantics of the resource. In the case of shared data structures, a more relaxed requirement suffices: execution of operations supported by the data structure, must have a result equivalent to some interleaving of these operations.

Solving a simple consensus problem among computing entities in a finite number of steps (i.e. in wait-free manner), requires special atomic operations [8]. Atomic compare-&-swap operations between private and shared registers (equivalent to processor registers and shared memory respectively), as well as some other operations on shared registers or FIFO queues, have been shown to solve the problem for an arbitrary number of threads [69]. Shared memory multiprocessor systems usually provide compare-&-swap, or weaker read-modify-write atomic operations in hardware for this purpose, and because such atomic operations cannot be constructed from simple memory reads and writes [70, 69].

Coming back to the first case, when synchronization is required to resolve a dependence, the completion of a read or a write that resolves the dependence needs to be propagated to the dependent thread. This propagation is done with a signal in hardware, in software, or both. When multiple such dependencies need to be satisfied, a thread can wait each one separately (distinct signal IDs or reception "places"), or count the relevant signals of resolved dependences up to the expected number (all signals arrive at the same place). The counter synchronization primitive proposed does the latter. In addition, multiple computations may depend on the completion of a read or write (or a set of reads and writes). In this case, multiple signals are required to "multicast" the event. Counter synchronization primitives support this need as well, by allowing multiple notifications, suitable for the construction of signal multicast trees. Two applications of the use of counters that demonstrate their versatility are presented next.

Figure 2.12 shows an example of RDMA-copy operation completion notification. Node A initiates a copy operation for 640 bytes residing locally, to be transferred to a destination scratchpad region that is scattered to nodes B and C (the destination region crosses a page boundary and the two pages reside in the different nodes). The acknowledgment address of the copy operation points to a pre-configured counter at node B, initialized at zero. The transfer is broken in five 128-byte packets, three of them sent to node B and two to node C. As part of the RDMA initiation, node A writes to the counter the opposite of the total transfer size (-640); this can be done asynchronously to the progress of the RDMA data transfer, but needs to be done through a single operation (e.g. it would be unsafe to first write -128 and later write -512).

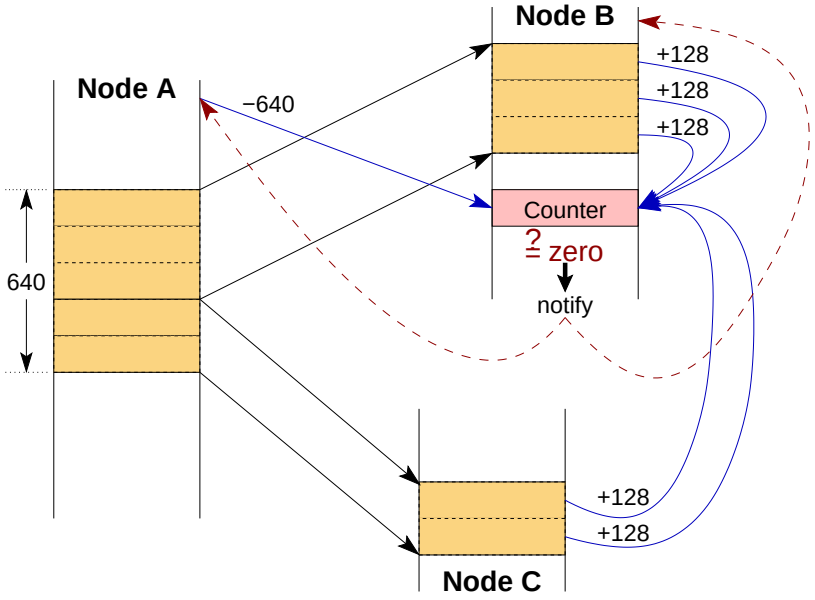


Figure 2.12: RDMA-copy operation completion notification.

For each packet of the copy operation that arrives at its destination, an acknowledgment is generated, writing the value 128 to the counter (remember that the data of the acknowledgement is the size of the written data). The counter accumulates the values of these acknowledgments, plus the opposite of the total transfer size sent by node A, and will become zero only once, when all of them have arrived. When that happens, the counter sends notifications to the pre-configured addresses, which in the example reside at nodes A and B.

Several points are evident in this example. First, observe that this completion notification mechanism does not require network ordering –the order in which the values are accumulated in the counter is not important. Second, the RDMA-copy operation of the example could be initiated equivalently by node B, or C, or by a fourth node instead of node A. The only difference would be the additional read request packet sent from the initiating node to node A before the illustrated communication is triggered[15]. This means that the completion notification mechanism is suitable for the general semantics of the RDMA-copy communication mechanism, and is not restricted to simpler *get* and *put* operations, that necessarily involve either source or destination data residing at the initiating node. Finally, observe that completion notification for an arbitrary number of user-selected copy operations can use a single counter, provided that the opposite of the aggregate size of all transfers is

[15]The write of the opposite of the total transfer size (-640) can actually be done by any node.

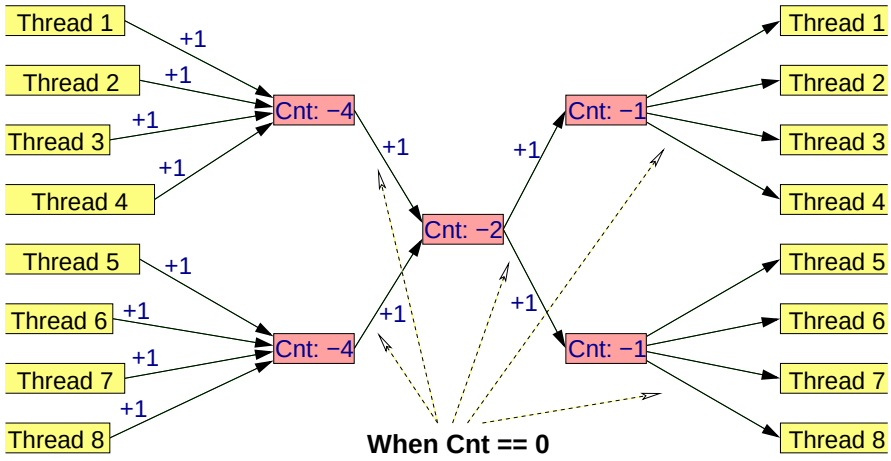


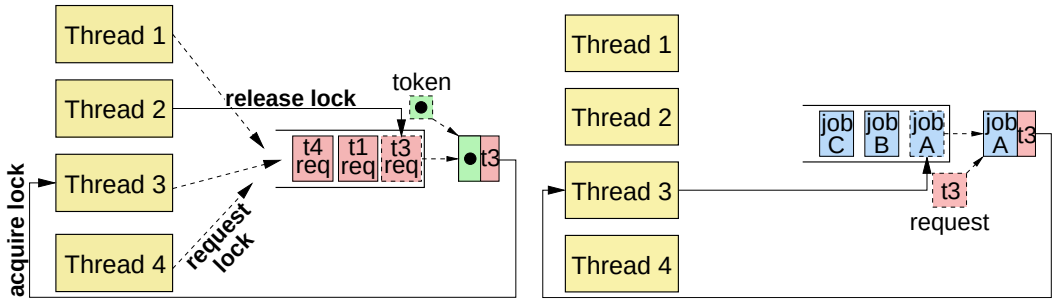
Figure 2.13: Hierarchical barrier constructed with counters.

written to the counter with a single operation, to prevent notification triggering for subsets of the transferred data.

Counters can also be used to construct scalable and efficient hierarchical barriers for global synchronization. Figure 2.13 shows how counters are combined to form two trees, with a single counter at their common root, in a barrier for eight threads. The tree on the left accumulates arrivals, and the tree on the right broadcasts the barrier completion signal. In the figure, threads are shown to enter the barrier writing the value one to counters at the leafs of the arrival tree. The counters of both trees are initialized to the opposite of the number of expected inputs. For the arrival tree, when counters become zero they send a single notification with the value one, that is propagated similarly towards the root of the tree. When the root counter triggers, the barrier has been reached. Counters in the broadcast tree, other than the root one, expect only a single input and generate multiple notifications, propagating the barrier completion event to the next tree level, until the final notifications are delivered to all the threads. Counters that trigger, sending notification(s), are automatically re-initialized and are ready for the next barrier episode.

When synchronization is required to control the access of shared resources, the multiple-reader queue can provide such coordination of competing threads reducing the cost of synchronization. This is shown in the following examples, by demonstrating efficient lock and job-dispatching services using a multiple-reader queue.

In figure 2.14(a) a multiple-reader queue (mr-Q) is used to provide a traditional lock. Initially, a single lock token is enqueued in the mr-Q as shown in the figure. The first dequeue (read) request acquires the lock, which is automatically forwarded



(a) Serve (lock) requests **(b) Distribute data (jobs)**

Figure 2.14: Multiple-reader queue provides a lock service by queueing incoming requests (a), and a job dispatching service by queueing data (b).

to the requester. Further dequeue requests are buffered in the mr-Q in FIFO order. The lock should then be returned (write) to the queue after each use, to be forwarded to other requesters either pending or to arrive. A possible generalization is to introduce $n > 1$ tokens in this system. In this case, at most n tasks are ever allowed simultaneously in what could be called a semi-critical section, providing a way to reduce contention for the resources accessed therein. The lock function exploits a multiple-reader queue as a hardware *binary semaphore* [68] with a FIFO queue of pending requesters, and requires scratchpad memory space on the order of the maximum possible number of simultaneous contenders.

Figure 2.14(b) shows how to implement a job dispatching system using a multiple-reader queue. Job-servicing threads, when idle, post work requests to the queue (read from the queue). In the scenario presented, previously posted (written) job descriptions are waiting in the mr-Q, to be matched with work requests. Multiple client threads (not shown) may be posting the jobs, and job-servicing threads do not need to interact or synchronize with them or with each other. Synchronization for matching jobs with service requests is done within the network interface hosting the mr-Q, implicitly. The converse scenario is also possible, where potentially multiple job-servicing threads send requests (reads) before any client thread posts (writes) a job. The cost of synchronization between servicing and client threads is amortized over multiple round-trips to the mr-Q, depending on the flow control scheme, the size of the queue, and the number of threads (see subsection 2.4.1). Thus, a multiple-reader queue can efficiently decouple and synchronize the threads and support job dispatching.

2.5 Related Work

In the following, work related to cache-integrated NIs is discussed, broken down in different research domains addressing previous research.

Coherent Shared Memory Optimizations. A large suit of communication mechanisms has been compared for producer-consumer patterns in [71], including prefetching, deliver [41], write-send [40], update-based coherence, selective updates [42], cache-based locks [72, 73], and a cache-based message passing scheme called streamline [74]. Streamline is found to perform best on benchmarks with regular communication patterns. Nevertheless, prefetching is the least expensive to implement and has good average performance on the 64 processor system they study, even under small variations to latency, bandwidth, and topology. Cache-integration of the network interface allows mixed use of coherent and direct transfers, shared access to scratchpad memory, and RDMA bulk communication that were not supported by Streamline and were not otherwise evaluated in [71].

In comparison with coherence-based producer-initiated transfers, such as deliver [41] and write-send [40], direct communication mechanisms should provide the qualitative advantages identified in subsection 2.2.2. Quantitative evaluation of execution overhead, as well as energy and power advantages, over normal coherent caching and prefetching, is provided in [75].

Network Interface Placement. Network interface (NI) placement in the memory hierarchy has been explored in the 90's. The Alewife multiprocessor [76] explored an NI design on the L1 cache bus, to exploit its efficiency for both coherent shared memory and message passing traffic. The mechanisms developed exploited NI resource overflow to main memory. At about the same time, the Flash multiprocessor [60] was designed with the NI on the memory bus for the same purposes. One of the research efforts within the Flash project explored block-transfer IPC implemented on the MAGIC controller (the NI and directory controller of a Flash node), and found very limited performance benefits for shared memory parallel workloads [77]. Generally, efforts in both these projects had limited, if any, success in demonstrating considerable performance advantages for message passing [78, 79, 77]. Later in the 90's, Mukherjee et al. [35] demonstrated highly efficient messaging IPC with processor cacheable buffers of a coherent NI on the memory bus of a multiprocessor node. The memory bus placed NI was considered less intrusive, and thus easier to implement, than the top performing cache-bus placed NI of that study, whose latency and banwidth advantages faded in application-based evaluation.

This body of research in the 90's explored less tightly-coupled systems than

those of today, and far less than future many-core CMPs. In addition, processor-memory speed gap was smaller in these systems. Cache-integrated NIs enable the opportunity for on-chip communication in large-scale CMPs, and exploration of alternative programming models than message passing. During the last decade, the abundance of available transistors transforms on-chip integration to an advantage, as demonstrated by chips targeting scalable multiprocessors [80, 81, 82], and more recent server chips like Opteron-based AMD CMPs and SUN Niagara I and II processors, which integrated the network interface on-chip. For manycores, further integration is justified, to reduce communication overheads, and to increase flexibility and programmability. Cache-integration of the NI moderates the associated area overhead.

NI Address Translation and User-level Access. Two other subjects, extensively researched in the 90's, concerned NI user-level access to overcome operating system overheads to communication [83, 84], and address translation in network interfaces to leverage their use for DMA directly from the application [60, 61, 62]. Despite extensive study and awareness of these performance barriers, the solutions proposed have not been adopted by operating systems and have been used only in custom high performance system area networks (e.g. [65, 67, 82]). Cache-integrated NIs leverage past research, and exploit close coupling with the processor to enable reuse of its address translation and protection mechanisms.

Configurable Memory and Event Handling Support. Ranganathan et al. [85] propose associativity-based partitioning and overlapped wide-tag partitioning of caches for software-managed partitions (among other uses). Associativity-based partitioning allows independent, per way addressing, while overlapped wide-tag partitioning adds configurable associativity. PowerPCs allow locking caches (misses do not allocate a line) (e.g. [86]). Intel's Xscale microarchitecture allows per line locking for virtual address regions either backed by main memory or not [87].

In smart memories [88] the first level of the hierarchy is reconfigurable and composed of 'mats' of memory blocks and programmable control logic. This enables several memory organizations, ranging from caches that may support coherence or transactions, to streaming register files and scratchpads. Their design exploits the throughput targeted processor tiles to hide increased latencies because of reconfigurability. It should be possible to support coherent cache and scratchpad organizations simultaneously and microcode-program smart memories for software-configurable actions. SiCortex [82] ICE9 chip features microcode-programmable tasks in a coherent DMA engine side-by-side with a shared L2 cache, but does not support scratchpad memory.

Completion queues have been proposed in the VIA [39]. Fine-grain access

control [89] demonstrates how lookup mechanisms leverage local or remote handling of coherence events. Exposing this functionality in Typhoon [90] allowed application-specific coherence protocols [91]. The work on fine-grain access control has influenced our approach to cache-integration of event responses (see subsection 3.1.3).

Our design generalizes the use of line state to support configurable communication initiation and atomic operations, in addition to fine-grain cache line locking that prevents scratchpad line replacement. With the cache integrated NI architecture, event responses utilize existing cache access control functionality, to enable modified memory access semantics that support atomic operation off-loading and automatic event notifications. Instead of a programmable controller or microcode, we provide run-time configurable hardware that can be exploited by libraries, compilers, optimizers, or the application programmer. Configurable memory organizations, such as smart memories, should incur higher area overhead compared to our integrated approach, although a direct comparison would require porting our FPGA prototype to an ASIC flow (the work on smart memories only provides estimates of silicon area for an ASIC process). Smart memories and ICE9 DMA engine are the closest to our cache-integrated NI, but our work focuses on keeping the NI simple enough to integrate with a high performance cache.

Hardware Support for Streaming. Recently with the appearance of the IBM Cell processor [12] design, which is based on separately addressable scratchpad memories for its synergistic processing elements, there was renewed interest for the streaming programming paradigm. Streaming support for general purpose systems exploiting caches for streaming data was considered in [92, 93, 45, 94]. Implementations side-by-side with caches or in-cache using the side effects of cache control bits in existing commercial processors are exploited in these studies. In [45] communication initiation and synchronization are considered important for high frequency streaming, whereas transfer delay can be overlapped providing sufficient buffering. Coherence-based producer-initiated transfers that deliver data in L2 caches, are augmented with a write-combining buffer that provides a configurable granularity for automatic flushing. Addition of synchronization counters in L2 caches (which they do not describe), and dedicated receive-side storage in a separate address space, increases performance to that of heavyweight hardware support at all system levels. In [92] a scatter-gather controller in the L2 cache accesses off-chip memory and exploiting in-cache control bits for best-effort avoidance of replacements. A large number of miss status holding registers enables exploitation of the full memory system bandwidth. Streamware [93] exploits the compiler to avoid replacements of streaming data mapped to processor caches, for codes amenable to stream process-

ing.

Stream processors like Imagine [50] and the FT64 scientific stream accelerator, provide stream register files (SRF) that replace caches, for efficient mapping of statically identified access streams. A SIMD computation engine pipelines accesses to the SRF, and bulk transfers between the SRF and main memory utilize independent hardware and are orchestrated under compiler control. Syncretic adaptive memory (SAM) [95] for the FT64 integrates a stream register file with a cache and uses cache tags to identify segments of generalized streams. It also integrates a compiler managed "translation" mechanism to map program stream addresses to cache and main memory locations.

Leverich et al. [96] provide a detailed comparison of caching-only versus partitioned cache-scratchpad on-chip memory systems for medium-scale CMPs, up to 16 cores. They find that hardware prefetching and non-allocating store operation optimizations in the caching-only system, eliminate any advantages in the mixed environment. We believe their results are due to focus to and from 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.

The Cell processor and [45] do not exploit dynamic sharing of cache and streaming on-chip storage, enabled by cache-integrated NIs. Support proposed in [92] can only serve streaming from and to main memory, although our architecture is likely to benefit from scatter-gather extentions for RDMA, possibly at the cost of more complex hardware. Both [45] and [92] forgo the advantages of direct transfers, provided by our design, and evaluate bus based systems. In addition, bandwith of cache-based transfers, used in these studies, is limited by the number of supported miss status holding registers (MSHRs) and the round-trip latency of per transfer acknowledgments, unlike the large number and arbitrary size RDMA transfers that can be supported, with lower complexity hardware, in cache-integrated NIs. Compiler support like that of Streamware is much easier to exploit with configurable cache-scratchpad. Stream register files and SAM, compared to our cache-integrated NI, require a specialized compiler and target only data-level and single instruction stream (SIMD) parallelism.

Queues Support for Producer-Consumer Communication Efficiency. Queues tightly-coupled to the processor have been proposed, among others, in the multi-ALU processor (MAP) [97], and in Tilera's TILE64 chip [7]. MAP's multithreaded clusters (processors) support direct message transmission and reception from registers. In addition, [97] demonstrates how multiple hardware threads in a MAP cluster, can handle in software asynchronous events (like message arrival or memory sys-

tem exceptions) posted in a dedicated hardware queue. The TILE64 chip [7] allows operand exchange via registers. A small set of queues are associated with registers, and provide settable tags that are matched against sender-supplied message tags. A catch-all queue is also provided for unmatched messages. These queues can be drained and refilled to and from off-chip memory.

Hardware support for software-exposed queues has been proposed, among others, for the Cray T3E [37], and in the remote queues [34] proposed by Brewer et al. Cray T3E [37] provides queues of arbitrary size and number in memory, accessible at user-level, that provide automatic multiplexing in hardware and a threshold for interrupt-based overflow handling. Remote queues [34], demonstrate three different implementations (on Intel Paragon, MIT Alewife and Cray T3D), exploiting in different degrees polling and selective interrupts. Remote queues demonstrate an abstraction that virtualizes the incoming network interface queue which may trigger a context switch on message arrival to drain the network as in active messages [53], or alternatively exploit buffering to postpone its handling. Two case delivery [52] in FUGU (modified Alewife), first demonstrated buffering in the application's virtual memory and enables virtualization of remote queues.

Virtualized single-reader queues in scratchpad memory (see subsection 2.4.1) enable direct transfer reception, that shares the fast and usual path through the processor's cache with coherent shared memory traffic, without occupying processor registers. Cyclic buffering is enabled by updating a hardware head pointer and multiple item granularities are supported, for efficient use of the limited on-chip space. Multiple single-reader queues can be allocated to demultiplex independed traffic categories, limited only by the available on-chip space. Flow control may be needed only at the user-level, and uses efficient on-chip explicit acknowledgements instead of interrupts. Selective user-level interrupts can be combined with single-reader queues to increase the efficiency of irregular or unexpected event handling, like operand or short message processing, although we do not implement or evaluate such support.

3

The SARC Network Interface

3.1 SARC Network Interface Architecture

The SARC network interface was designed and implemented within the SARC [55] European Integrated Project (FP6 IP #27648). The whole project aimed to provide a multidisciplinary framework for scalable computer architecture, that included the design of architecture, system software, interconnection networks, and programming models, to address the challenges of chip multiprocessor and system-on-chip technologies, and of upcoming application domains. The SARC network interface demonstrates cache integration of the NI and supports scalable communication and synchronization mechanisms for a global address space.

Cache integration of the network interface provides user-level sharing of processor's memory for communication purposes, and memory hierarchy resource sharing for implicit and explicit communication mechanisms. Explicit, acknowledged transfers, via load-store, message, and RDMA-copy operations, support direct communication. Command buffers for multi-word communication initiation, counters for completions notification, and queues for multi-party synchronization, complement the architectural framework for explicit communication. To enhance scalability of the virtual address space in the presence of data migrations, progressive address translation is introduced. These are discussed in the following subsections.

3.1.1 Large Region Protection and Progressive Address Translation

Progressive address translation separates protection from address translation. An *address region table (ART)* supports protection for large virtual regions at communication source nodes, and abstract locality information in the form of a local or re-

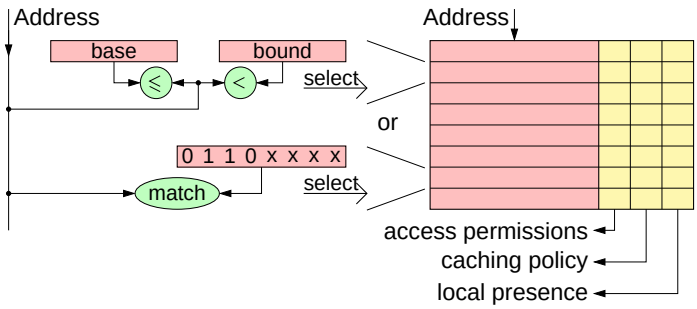


Figure 3.1: Address Region Tables (ART): address check (left); possible contents, depending on location (right).

mote indication. In addition, to support both coherently cacheable and non-coherent regions, ART also provides region type information. To avoid aliases of cacheable address regions, a single virtual address space is proposed [1]. A small number of entries is required, and ideally ART accesses will never miss.

Figure 3.1 shows the functionality provided by address region tables and some possible region identification methods. Regions with arbitrary base and bound can be supported, e.g. with two parallel comparisons per table entry, or power of two address aligned regions, e.g. specifying in each entry a variable number of "don't care" bits [2]. ART entries specify access permissions, cacheability, and local presence attributes for each region. The caching policy can be used to indicate whether a region is cached, if so in which hierarchy levels, as well as write-through, write-back, and write-allocate options. Similarly, pinned or migratable properties can be associated with a region.

For progressive address translation, proximity domains are supported. Instead of single-step address translation, intra-domain routing tables are included in cache-integrated network interfaces and inter-domain routing tables at domain boundaries, in a search for communication destinations that may have multiple steps when across locality domains. Progressive address translation reduces the "size" of translations identifying destination nodes or node groupings, instead of a physical address. For a more detailed description of progressive address translation refer to [63].

3.1.2 User-level Access and Virtualized NI "Registers"

As discussed in section 2.4, direct communication mechanisms rely on an address space extension that accommodates scratchpad memories (refer to figure 1.3). Protected access in the global address space is steered by the address region table as illustrated in figure 3.2. Other than verifying protection rights for accesses, ART indicates whether an access refers to a cacheable or a directly addressable address region, and in the later case if the scratchpad region is local or remote. The union of all directly addressable regions, comprise the non-coherent address space.

Global virtual address bits are used for cache indexing of scratchpad regions. The local presence properties provided in ART entries (see figure 3.1) may include a

[1] Alternatively, parallel ART and TLB access can be used, which avoids the need for virtual caches.

[2] Other possibilities for region identification include pages and super-pages as provided in modern TLBs, and domain identifiers in the virtual address as in the Single-chip Cloud Computing (SCC) Intel research chip [5].

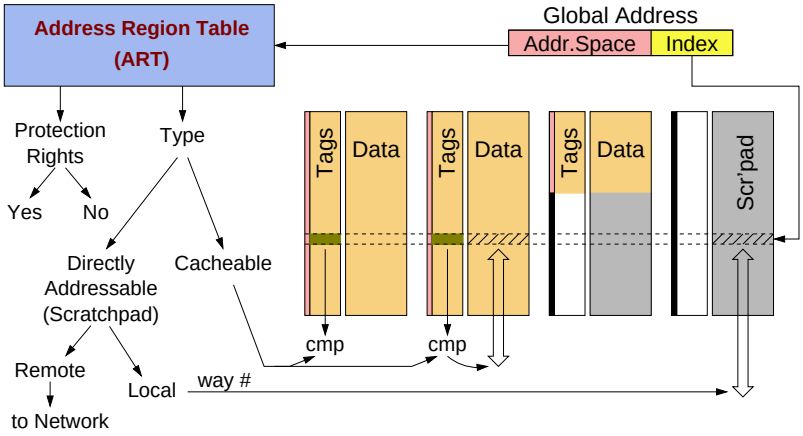


Figure 3.2: Memory access flow: identify scratchpad regions via the ART instead of tag matching –the tags of the scratchpad areas are left unused.

specific cache way number for local scratchpad regions. Otherwise, the scratchpad virtual address must provide these bits, too[3]. By discriminating among cacheable, local scratchpad, and remote scratchpad regions, ART allows the cache to select the appropriate addressing strategy and avoid miss status holding register (MSHR) and relevant space allocation for remote scratchpad accesses. Because scratchpad regions are identified in the ART, tag matching is not required in the cache. The tags of scratchpad lines are "freed" and are used for network interface metadata, as discussed below.

The cache-integrated network interface exploits cache state bits for two purposes: (a) to allow scratchpad region allocation at cache-line granularity, and (b) to provide alternative access semantics inside srcatchpad regions. The use of per line state bits to designate scratchpad areas inside the cache, allows flexible management of on-chip memory usage for different purposes circumventing the coarse protection granularity. Furthermore, the cache-integrated NI uses the state bits to support communication and synchronization primitives implemented inside the scratchpad address space, as depicted in figure 3.3. In addition to the above, either ART identification of directly addressable regions, or cache state bits can be used to prevent cache tag matching for scratchpad memory and other special scratchpad lines discussed below. State bits are preferable, since they obviate ART lookup at a remote NI, or the need for NoC packet overhead.

Five different lines types are shown: i) normal cached data lines marked (Ch); ii) normal scratchpad memory lines marked (LM); iii) Command buffers (commu-

[3]Alternatively, ART could provide low order bits of the physical address, e.g. at page granularity, for intra-node addressing (i.e. in local regions).

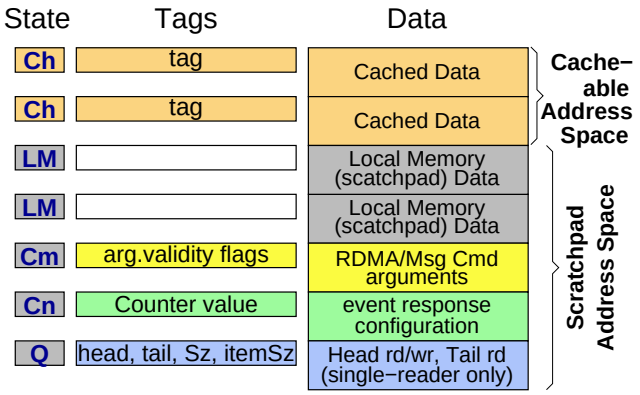


Figure 3.3: State bits mark lines with varying access semantics in the SARC cache-integrated network interface. Cacheable and scratchpad address spaces are identified via ART access.

nication control and status) marked (Cm); iv) counter synchronization primitives marked (Cn); v) single- and multiple-reader queue descriptor synchronization primitives marked (Q). Counters and queues provide the direct synchronization mechanisms presented in chapter 2 (subsection 2.4.1). Queue descriptors correspond to the queue control block discussed there. Details on the functionality of the communication and synchronization primitives are discussed in the next subsection. The full description of their implementation, as well as the format and use of the tag and data block for each will be provided in subsection 3.2.5.

Along with command buffers, counter and queue descriptor lines are the equivalent of NI registers and are called *event sensitive lines (ESLs)*. Multiple ESLs can be allocated in the same or in different software threads. Since ESLs are placed inside the scratchpad address space, ART provides access protection for them. Each software thread can directly access at user-level ESLs (and scratchpad memory), independently of and asynchronously to other threads. For protection checks of communication address arguments, an additional ART must be provided in the network interface. Migration and swap out of a scratchpad region must be handled by the OS, as discussed in the previous chapter (section 2.3.3).

The network interface keeps track of multiple outstanding transfers, initiated via the communication and synchronization primitives, using a job-list. Although a linked list could be implemented to allow an almost unlimited number of outstanding transfers, this would entail multiple next pointer accesses in the cache tags (or data), and would consume cache bandwidth and energy. In our implementation we use a fixed size external FIFO for the job-list (see subsection 3.1.4), that allows job description recycling to overlap implicit and explicit transfers and multiple inde-

pendent bulk transfers. The fixed job-list size restricts the total number of ESLs and concurrent transfers that can be supported, but it can be scaled with low hardware complexity increase.

The use of state bits obviates the need of a network interface table for ESLs and circumvents protection granularity, solving two of the problems phased with the connection-oriented approach of subsection 2.1.3. Avoiding the need for a table keeping state for NI outstanding transfers, also requires the explicit acknowledgements of subsection 2.4.2, which are also part of the SARC architecture.

3.1.3 Event Responses: Cache Integration of the Network Interface

Event responses is a technique that exploits tagged memory to enable software configurable functions, extending the usual transparent cache operation flow, in which line state and tag lookup guides miss handling with coherence actions. Local or remote accesses to ESLs (NI events) are monitored, to atomically update associated NI metadata. Conditions can be evaluated on NI metadata, depending on the event type (e.g. read or write), to associate the effect of groups of accesses with communication initiation, or to manage access buffering in custom ways. These conditional NI actions are called event responses, and allow the implementation of different memory access semantics. The mechanisms designed, utilize appropriate ESL and scratchpad region internal organization, to buffer concurrent accesses and automatically initiate communication.

Event responses provide a framework for hardware synchronization mechanisms configurable by software, which allows the implementation of the direct synchronization mechanisms introduced in chapter 2 (subsection 2.4.1). In addition, they allow the description of multi-word communication arguments to the NI and automatic transfer initiation, enabling the NI communication functions. The different operations are designated by ESL state corresponding to the intended access semantics.

On every access to the cache (local or remote), normal cache operation checks the state and tag bits of the addressed line. The NI monitors ESL access, reads and updates associated metadata stored in the ESL tag, and checks whether relevant conditions are met, as illustrated in figure 3.4. The state accumulated in the ESL tag can be used to customize the location of buffering, forming for example queues in scratchpad memory. When communication triggerring conditions are met, communication is scheduled by enqueueing a job description in the network interface job

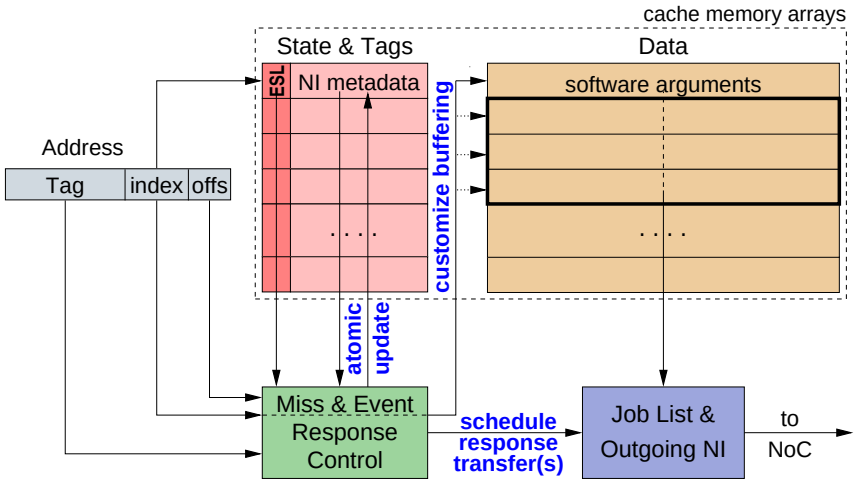


Figure 3.4: Event response mechanism integration in the normal cache access flow. ESL accesses are monitored and check of NI metadata in the ESL tag can be used for custom access buffering in scratchpad memory (e.g. managing some lines as a queue), or to schedule response transfer(s). When, later, the outgoing traffic NI controller initiates such transfer(s), it can find related software arguments in the ESL data block.

list (see section 2.3.3). Software communication arguments can be pre-configured in the ESL data block. Such arguments are read and utilized by the outgoing traffic NI controller (NI out for short), when the relevant job description appears at the top of the job-list.

The operation of event responses closely matches normal cache operation, with two exceptions: (i) customized buffering requires that the tag and data arrays are accessed in different cycles, and (ii) the NI job list is not necessary for miss request processing and may introduce queueing latency to miss handling. Nevertheless, customized buffering fits well with the operation of phased, pipelined caches that are usual at the second level of the memory hierarchy, and prevent accessing all the associative data arrays for each search saving energy. In case cache operation requires parallel tag and data array access, custom buffering would introduce a bubble in the pipeline, increasing the relevant operation delay. The additional cache miss latency introduced by the NI job list can be managed by separately handling miss requests, bypassing the job list. Thus event responses provide the means for network interface cache integration, exploiting functionality reuse to a large degree.

Four event-response mechanisms are designed, for command buffers, counters, single- and multiple-reader queues. Command buffers are used to send messages or initiate RDMA-copy operations. An opcode, size, destination and acknowledgement addresses are provided for messages, and data that may add up to the cache line

size. In the case of RDMA-copy operations, a source address is provided instead of the data. The acknowledgement address is utilized for explicit acknowledgements as described in chapter 2 (subsection 2.4.2). Command buffers exploit operation opcode and message size to *automatically* initiate a transfer once its description is complete –assuming all transfer arguments are written to a command buffer exactly *once*.

Command completion is monitored at word granularity, allowing arbitrary order of the relevant stores that can be separated by any time distance. The ability to tolerate arbitrary ordering of stores describing transfer offloading, allows compiler reordering of instructions and store issue optimizations for weakly ordered memory accesses. In addition, the appropriate access rights for address arguments are also monitored for transfer initiation, utilizing access of the ART for each store. The automatic detection of a complete transfer description obviates restrictions related to load and store reordering and the need for an additional access, required for explicit initiation. Multiple threads can be writing command buffers within their protection domain, concurrently, in user-mode.

Counters support atomic add-on-store and up to four configurable notification addresses, as described in chapter 2. The notification addresses, a single word for notification data and a counter reset value can be configured in advance in the ESL. When the counter reaches zero, four stores of the notification word are scheduled toward the notification addresses, by enqueueing corresponding job descriptions in the network interface job list. In addition, the counter is reset to the pre-configured value.

Single- and multiple-reader queues keep head and tail pointers in the ESL tag. The queue is formed in scratchpad memory adjacent to the ESL. The multiple-reader queue actually implements a pair of overlapped queues, one for read/dequeue requests and one for written/enqueued messages, and thus an implementation requires three pointers (e.g. a single head and two tail pointers: reads-tail and writes-tail), to keep track of matched entries for which a generated dequeue response is in progress. Tail pointers guide customized buffering by modifying some of the least significant index bits, to access the pointed queue entry in the data array; this is shown in figure 3.4. For the multiple-reader queue, matching of a read with a write in the queue is associated with a job description placed in the NI job list (the exact implementation is discussed in subsection 3.2.5).

Single-reader queue writes, as well as multiple-reader queue reads and writes, increment and update tail pointers in the ESL tags *atomically*, unless the queue is full. In the latter case, the NI either updates a receiving node error indicator and generates an interrupt, or sends a NACK to the acknowledgement address of the

superfluous read or write[4]. In both cases the incoming packet is dropped.

Processing from NI-out of a read-write match in a multiple-reader queue, utilizes the head pointer to read response message information. When the response is sent, the queue's head pointer is also atomically incremented and updated in the ESL tags. The single-reader queue does not trigger communication, other than acknowledgements generated for all remote writes (see chapter 2, subsection 2.4.2); its head pointer is updated according to local stores at a specific ESL offset.

3.1.4 The Network Interface Job List

In order for a network interface integrated at top memory hierarchy levels to manage multiple pending operations, it must be able to follow of their status until they are complete. A FIFO queue of initiated transfers is suitable for hardware implementation. Such a queue can be implemented either as a linked list of outstanding transfers, linking together ESLs to keep track of communication state, or as a separate FIFO that stores job descriptions pointing to ESLs when necessary. As discussed in subsection 3.1.2, using a linked list of ESLs can result in multiple accesses to next pointers, wasting cache bandwidth and energy, so in the following we concentrate to a job-list of fixed size as the one implemented in the SARC prototype.

Because of NI close coupling with the processor, it is important to allow processor demand requests to bypass ongoing bulk transfers (i.e. RDMA-copy operations). This is significant in order to allow local computation, that may depend on remote accesses, to proceed in parallel with bulk, offloaded communication. Furthermore, because processor remote accesses generate transfers of fixed, short size, bypassing bulk transfers in this case enhances the pipelining of short ones. The resulting latency overlap of short transfers effectively hides remote access latency, improving computation efficiency.

Moreover, it may be profitable to "prioritize" mechanisms for short transfers as urgent, allowing all short transfers to bypass RDMA operations. Such optimization of short transfers for low latency communication and bulk ones for throughput is meaningful because the transfer latency "seen" by software is commensurate to its size, since the program normally waits for transfer completion. As a result, tailoring mechanisms to different aspects of communication efficiency, allows software to exploit the most appropriate operations for the task at hand. This will be demonstrated in the evaluation of the hardware prototype in chapter 4 (section 4.1). Note

[4]The NACK may also causes an interrupt at its destination. In all cases, such interrupts can be implemented as privileged or user-level. The SARC prototype does not implement such interrupts.

that all the communication mechanisms of the cache-integrated NI except RDMA, require single packet transfers, either for short requests and acknowledgements, or for writing data of up to one cache line.

In order to provide such bypassing, the job list can *recycle* partially executed (in-progress) RDMA-copy operations, when other operations are pending. Recycling, though, must ensure all operations make progress, so that bulk transfers are not starved during periods that many short jobs appear because of processor accesses and incoming traffic. Thus, the outgoing traffic controller should serve each operation that appears at the top of the job list, although RDMA-write operations need to be processed incrementally, breaking them into segments and recycling unfinished operations to the tail of the job list. Job descriptions in the job list *should not* be recycled without serving at least a single job (fully or partially), e.g. with the intention to avoid head-of-line blocking. Such recycling does not provide progress guarantees, and may rarely lead to livelock. In addition, it can increase performance variance, because it allows external traffic to interfere in unexpected ways with a node's operation.

At least three independent (sub)networks are necessary to support coherence[5]. To support the access semantics of subsection 2.4.2, explicit transfers generate read, write, and acknowledgement packets that can be also spread in three subnetworks. Only two are really necessary for explicit transfers in SARC architecture, since we limit the number of concurrent reads from each node to a software configurable maximum number for which the node provides buffering. Service of read requests is provided via a customized multiple-reader queue. Because buffering for explicit read requests is provided in the special multiple-reader queue, the responses can be initiated when the subnetwork carrying write traffic is available, without occupying the read subnetwork in the meantime. In effect, an independent read subnetwork is redundant, since nodes always sink read packets putting them in the special multiple-reader queue, or dropping them if the queue is full. Nevertheless, SARC architecture utilizes at least three subnetworks in order to support implicit communication as well as explicit.

Unfortunately, the job list *cannot* serve in a single, monolithic queue the different traffic classes of one or more protocols that must use independent subnetworks (e.g. implemented with virtual channels), because this can lead to deadlock when the queue is filled with pending jobs. Such a deadlock scenario is shown in figure 3.5. Two nodes, have deadlocked while processing a bulk RDMA job each destined to the other (longer cycles are also possible). The RDMA operation of

[5]This is because of coherence protocol dependencies of the form INV–>ACK–>DATA or INV–>WriteBack–>DATA that involve three distinct controllers.

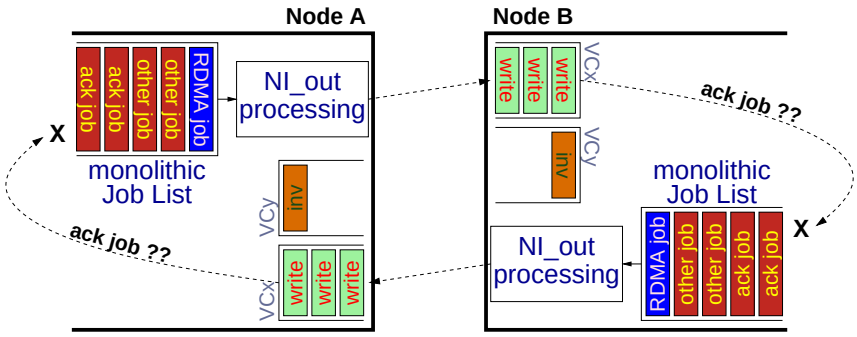


Figure 3.5: Deadlock scenario for job list serving multiple subnetworks.

each node cannot make progress because other write packets (independent or of the same operation) have filled all buffers in the path to the other node, which cannot sink the writes. Writes cannot proceed in either node because the corresponding acknowledgement cannot be enqueued in the job list which is full. This inability to sink a class of packets can arise periodically in any protocol, and does not cause deadlock when the "response" (the acknowledgement in this case) uses an independent (sub)network and is guaranteed to (eventually) make progress. This is not the case in the scenario shown, because the job list does not provide different FIFOs to support the independent (sub)networks. As a result, the job lists are filled with other jobs waiting behind the RDMA operation, and there is no space for acknowledgement jobs in response to the writes. Since the RDMA cannot make progress, the system is deadlocked.

Similarly, an invalidation request cannot be serviced in either node, since the corresponding acknowledgement job cannot be enqueued in the job list, which can involve other nodes in the deadlock. Note that the jobs behind the RDMA that are filling the job list may be previous write acknowledgements, and cannot be limited so that they fit the size of the job list. To prevent this situation, separate FIFOs for each independent subnetwork must be used. This would allow acknowledgements to proceed independently, and thus destination would eventually sink all writes. In effect, the job list should only recycle RDMA operation descriptions after minimal service, separate traffic classes, and possibly serve different traffic classes in round-robin.

Caches include queues with similarities to the job list, in order to manage the initiation of write-back operations. The queues used in the case of write-backs however, do not require operation recycling and may not need to arbitrate for the cache-scratchpad if data have moved to a victim buffer [98] at miss time. On the flip side, the job list can replace output packet buffers with sorter job descriptions, directly ar-

bitrating for the network when a packet can be constructed. This area optimization is straightforward when the local NoC switch operates in the same clock domain with the processors it connects, as implemented in our prototype. Otherwise, it may be complex to hide the synchronization cycles lost for accessing the cache-scratchpad allocator from the outgoing traffic NI controller and getting back the data in the network clock domain. In this case, small packet buffers per subnetwork will probably be beneficial.

Finally, it should be noted that scaling the size of the job list, results in small hardware complexity increase, because memory decoders and multiplexers have logarithmic time complexity and the latter can be overlapped with memory array access. Fixed partitioning of a single memory array can be used for queues per subnetwork, as implemented in our prototype.

3.2 SARC Network Interface Implementation

3.2.1 FPGA-based Hardware Prototype and Node Design

The SARC architecture, described in the previous section, was fully deployed with a hardware FPGA-based prototype implemented in the CARV laboratory of ICS-FORTH. The prototype utilizes a Xilinx Virtex-5 FPGA and development employed Xilinx EDK, ISE, and Chipscope tools. A previous version of the prototype was presented in [56]. The current version is a major rewrite of the code, optimized for logic reuse, including several features not present in the version of [56]. The prototype implements all the different memory access semantics of subsection 2.4.2, exploits event responses for the communication and synchronization primitives presented, and provides three levels of NoC priority to cope with protocol deadlock issues for both implicit and explicit communication mechanisms, that are also leveraged for explicit acknowledgements.

The block diagram of the FPGA system is presented in figure 3.6. There are four Xilinx IP microblaze cores, each with 4KB L1 instruction and data caches and a 64KB L2 data cache where our network interface mechanisms are integrated. An on-chip crossbar connects the 4 processor nodes through their L2 caches and NI's, the DRAM controller –to which we added a DMA engine– and the interface (L2 NI) to a future second-level, 3-plane, 16×16 interconnect, over 3 high speed RocketIO transceivers, to make a 64-processor system. All processors are directly connected over a bus (OPB) to a hardware lock-box supporting a limited number of locks.

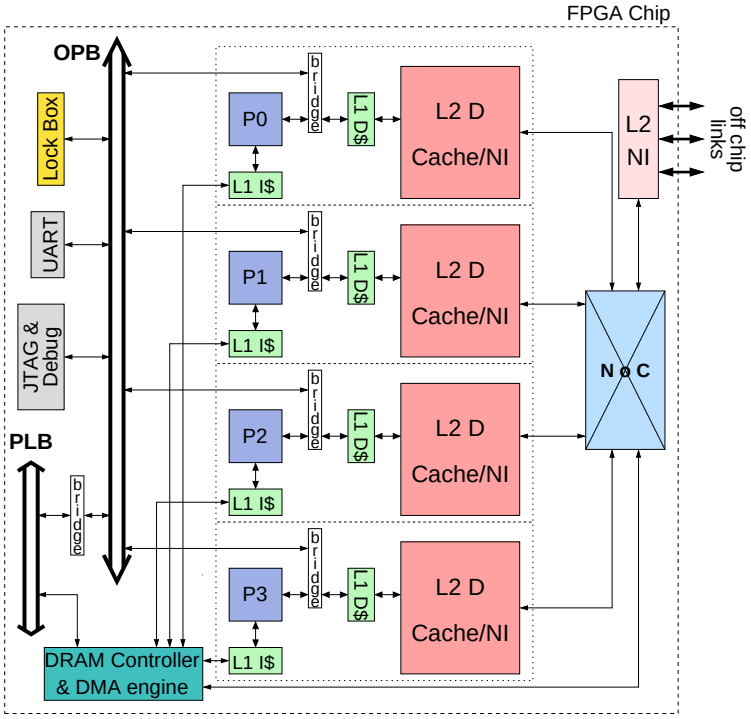


Figure 3.6: FPGA prototype system block diagram.

The lock-box is used in the evaluation of the hardware prototype for comparison purposes (see section 4.1).

The block diagram *does not* show the processor ART, accessed in parallel with the L1 data cache, and the store combining buffer, accessed in the subsequent cycle for remote stores. The prototype implements a network interface integrated at the L2 cache level of a private L2 cache. Cache integration at the L1 level could affect the tight L1 cache timing, because of the arbitration and multiplexing required among processor, network incoming, and network outgoing accesses to NI and cache memory. On the contrary, L2 cache integration mitigates timing constraints, provides adequate space for application data, and allows sufficient number of time overlapped transfers to hide latencies.

Integration with a private L2 has the advantage of independent, parallel access by each processor to NI mechanisms. It also allows the selective L1 caching of L2 scratchpad regions with minimal coherence support, using a write-through L1 cache and local L1 invalidation on remote writes to scratchpad; the SARC prototype implements this optimization. Coherence among the L2 caches of the four nodes to support implicit communication is under development at this time.

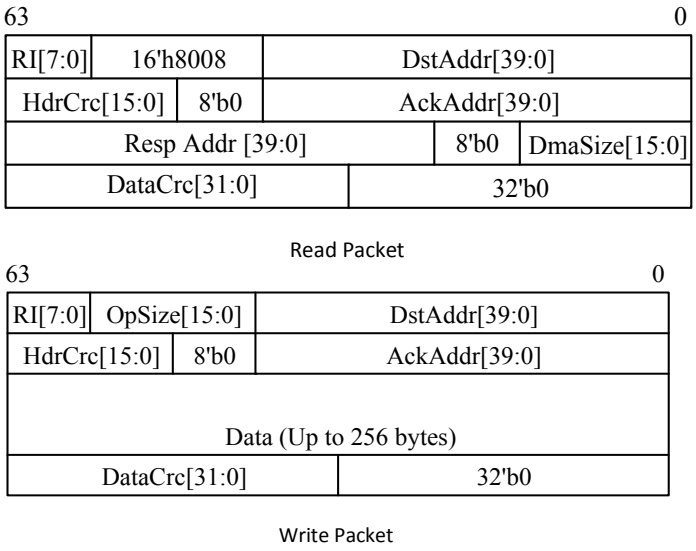


Figure 3.7: FPGA prototype system block diagram.

Finally, the L2 cache supports a single outstanding miss, which is useful only for cacheable stores that can be immediately acknowledged to the in-order IP core. The L1 does not implement a miss status holding register (MSHR), because non-blocking cache support is only used for a single store access that is immediately acknowledged to the processor. Extending the processor pipeline to include L1 and L2 data cache access and providing multiple MSHRs, can support limited cache miss pipelining, as will be discussed in the next subsection. The hardware prototype does not support these optimizations of cacheable accesses.

3.2.2 NoC Packet Format

Figure 3.7 shows read and write packet formats. These correspond to packets of all three subnetworks, that are implemented with NoC priorities in the prototype. Read packet format is used for requests (e.g. shared or exclusive coherence requests will probably use this format too). Write packet format is used for RDMA-write packets and acknowledgements –coherence data responses will also use this format. A packet is illustrated as a downwards succession of words of NoC width (64 bits), and bitfields are shown in verilog notation.

The upper two 64-bit words in either packet format are the header, which is the same for both read and write packets. In the first (topmost) word, from left to right, there are 8 bits of routing information (RI), followed by 16 bits for opcode

and packet size, and 40 bits for the destination address. All addresses are virtual, as required for progressive address translation. Although processors in the SARC prototype support only 32-bit addresses, 40-bit addresses are provisioned in the packet format for future extensions. The read packet format shows the exact opcode and size of read type packets for direct communication. In the second header word there are 16 bits of header CRC, 8 bits of zero padding, and 40 bits for the acknowledgement address.

In read type packets the header is followed by a single 64-bit word of payload, which comprises of 40 bits for a response address, 8 bits of zero padding, and 16 bits of request size labeled DMASize in the figure. In the case of write type packets, the header is followed by up to 256 bytes of payload (i.e. 32 words of 64 bits). Finally, for both packet types the payload is followed by one trailing word that includes a 32-bit CRC for the payload and 32 bits of zero padding. Subsection 3.2.6 describes optimization of these packet formats adapting to other features of the implementation, in order to reduce the area increase from network interface cache integration.

3.2.3 NI Datapath and Operation

Figure 3.8 shows the complete datapath of the SARC L2 cache-integrated network interface implementation. The datapath is separated in three controller domains: (a) the L2 cache and scratchpad controller domain, (b) the outgoing traffic controller domain, and (c) the incoming traffic controller domain. Four agents compete for access to the cache and scratchpad, arbitrated by the allocator module, which implements most of the control of domain (a). The processor interface has independent agents (i) for store combining buffer accesses, and (ii) for all L1 data cache misses and scratchpad region accesses except remote stores. The incoming and outgoing traffic controllers are the other two agents interfaced to the allocator.

The store combining buffer receives remote scratchpad stores after accessing the processor ART, that identifies remote destinations. Successive pipelined operation stages are shown in domain (a), but not in the other domains which require multiple arbitration cycles, and reuse their datapath for multiple network packet flits (words of NoC channel width). In addition, figure 3.8 hides some implementation details for presentation purposes. These include the single cache miss status holding register (MSHR) and registered data for a store miss. In addition, in the incoming traffic controller domain, the register shown is actually implemented with load enables per 32-bit word and previous multiplexing of replicated data is required to support sub-block granularity items for single-reader queues.

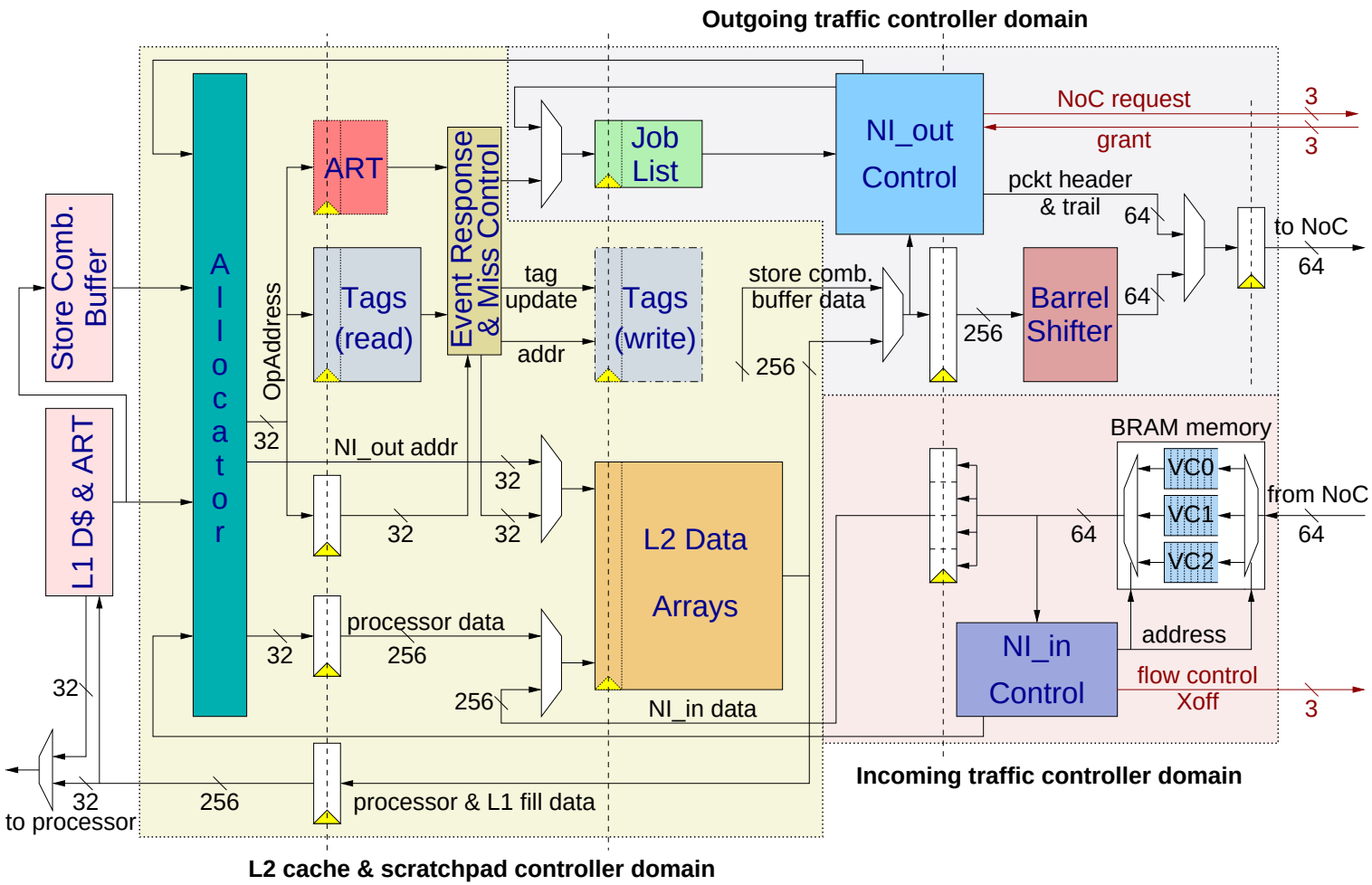


Figure 3.8: Complete datapath of SARC cache-integrated NI.

For common cacheable and local scratchpad accesses, the allocator is accessed after processor ART and L1 cache access but in the same cycle. After a grant from the allocator, the selected agent reads the tags and utilizes the network interface ART (separate from the processor ART). The current implementation has a minimal implementation of ART (providing region local or remote indication without protection), so ART is shown with a dotted outline. Event response conditions and tag matching are also evaluated in this cycle.

In the next cycle, depending on operation type, hit or miss calculation, and event response, data arrays may be accessed, the tags can be updated, and a job description can be enqueued in the network interface job-list. Non all operations write the tags (hence tag write is shown with a dash and dot outline), but for those that do, a pipeline bubble prevents subsequent operations from reading the tags and resolves the structural hazard. Domain (a) has priority in enqueueing jobs in the job-list over RDMA operation recycling, which may be delayed. For processor local scratchpad loads or cacheable L2 load hits, data are returned to the processor and L1[6] from an intermediate register, that adds a cycle of latency.

The outgoing traffic controller domain (or NI-out for short), processes jobs appearing at the top of the job-list. NI-out controller requests the appropriate NoC port and priority level based on routing table lookup and job type. The routing table is trivial in the current implementation, since only physical addresses are used (progressive address translation is not implemented). When a NoC request is granted, NI-out controller proceeds to retrieve data for the transfer, either from the store combining buffer or by requesting memory access from the allocator.

When a multi-word transfer is requested (message or RDMA), the associated command buffer is read from memory. For RDMA-write (i.e. an RDMA-copy operation with a local source), the appropriate data are requested subsequently. Command buffer information, described in subsection 3.2.5, allows NI-out controller to construct the packet header. The data are then sent to the NoC in successive cycles. Destination alignment is used supported with a barrel shifter. RDMA write packets are segmented to a maximum of four (4) cache lines of payload, depending on source and destination alignment. If the operation is not finished, it is recycled in the job-list, updating the command buffer appropriately. For read type packets the NI-out controller provides the single word of payload. The trail flit with the payload CRC is added in the end of the packet.

The incoming network interface controller domain (NI-in for short) processes packets arriving from the network. The packets are buffered in independent FIFOs

[6] The L1 is not filled for ESL accesses.

per network priority, implemented in a single SRAM block (BRAM). If the CRC of a packet's header is not correct, the packet is dropped. When a priority's FIFO is almost full, an xoff signal is raised to the NoC switch scheduler, to block the scheduling of further packets with this destination and priority until the condition is resolved. Packets are dequeued from the other end of FIFOs in priority order, as specified by the implicit and explicit communication protocols used [7] (this order matches the NoC priority order). In order to implement cut-through for incoming traffic, a write packet can be delivered to memory if its header CRC is correct, even though the data CRC can be wrong. In the later case, an error is recorded.

For correct packets, header information are processed by NI-in controller. When working on a write type packet, up to one cache line of data is placed in a register, as shown in the figure. For read packets the whole packet is stored in the register. In parallel with the dequeue of the last word of payload, NI-in controller issues a request to the allocator. When granted, the data in the register are appropriately buffered in the data arrays.

Virtual cut-through switching implemented for the prototype's NoC requires uninterrupted packet transfer. In order to support multiple cache lines of data in RDMA write packets, NI-out needs to have higher priority in the allocator. In addition, the allocator allows the outgoing traffic controller to bypass the tags read stage of the pipeline when desired, and directly read the data arrays. This may cause a pipeline bubble for an in-progress operation that would access the data in the cycle after NI-out is granted access[8]. If the in-progress operation does not need to access the data array (e.g. a cache miss), the bubble is not necessary.

The processor interface can utilize on average two out of every four cycles of the allocator, because NI in and out do not issue requests faster than one every four cycles. There is only one exception to this rule about the frequency of NI-in and NI-out requests to the allocator, that will be discussed in the following. The reason for the rule is that the data arrays have 4 times the width of the NoC (i.e. 256-bits versus 64-bits). In result, selecting NI-in or NI-out in the allocator consumes and provides data respectively with four times the bit rate of the datapaths of NI-in and NI-out. Giving higher priority to NI-out over other agents in the allocator does not affect NI-in, which also has available one out of every four cycles using the second highest priority in the allocator.

[7] This is not necessary. VCs could also be processed in round-robin.

[8] In the actual implementation there are two arbiters, one for the tags in two successive cycles as well as the job-list in the second, and one for the data. Thus, not only the current operation at the data stage suffers the bubble, but also any succeeding operation. The arbitration described here is equivalent, because other operations cannot be selected when NI-out is selected.

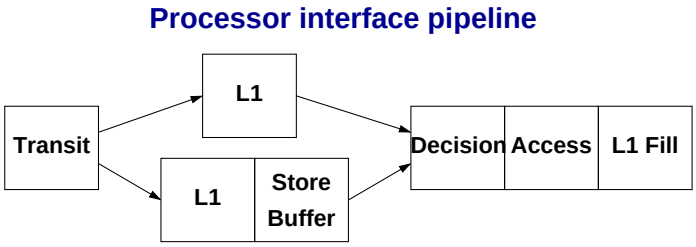


Figure 3.9: Pipelining of processor requests.

3.2.4 Processor Access Pipelining

The processor interface pipeline supports the following operation categories:

1. Processor cacheable and scratchpad loads;
2. Processor cacheable and scratchpad stores;
3. Processor ESL loads and stores.

Not all operations in each category access all the stages of the processor interface pipeline shown in figure 3.9. This may result in out-of-order completion of operations in the memory system (e.g. an L1 load hit can complete before a previous L2 store hit), which is correct for the weak memory ordering supported by the SARC prototype and architecture. From the perspective of the processor loads cause stalls and all stores are acknowledged immediately, as long as the pipeline to the L2 is not both filled with in progress stores and stalled[9].

Because the processor is an IP core the memory access address is supplied from a register, and thus requires a *Transit* cycle before the L1 is accessed –all FPGA SRAM blocks (BRAMs) require that the accessed address is ready at the beginning of the cycle. In the *L1* stage, the L1 cache is accessed in parallel with the processor ART. All operations except remote scratchpad stores, utilize the upper "path" of the pipeline that does not include the *Store Buffer* stage to access the single-block write combining buffer. In the *Decision* stage, the L2 tags are read and the network interface ART is accessed. The latter is only used to check the access rights for the data of stores in the third category, in case these are used as communication address arguments in command buffers and counters[10]. In addition, in this

[9] No consistency constraints are implemented at the moment on the prototype. As a result, user software must enforce the desired order between remote store accesses, by waiting for acknowledgements as necessary, and wait for all previous remote store completion before issuing a remote load. Completion detection of previously issued remote stores is supported with a special memory mapped NI register.

[10] Alternatively, a second port to the processor ART could be used for this purpose.

stage event response and miss control decides if customized buffering is dictated by the accessed memory semantics and whether an access will trigger communication accessing the job-list. In the *Access* stage, the data arrays are accessed, the tags may be written, and a job may be enqueued in the job-list. Finally, loads that hit in the L2 utilize an *L1 Fill* stage to update the L1 and deliver data to the processor.

Since local scratchpad memory is cacheable in the L1, operations in the first category (loads) may hit in the L1 cache, which completes their processing in the stage labeled *L1* returning the data to the processor in the same cycle. Loads that miss in the L1 cache, issue a request to the L2 cache allocator at the end of the *L1* stage, which will stall the pipeline if not granted immediately. After a grant from the allocator, a load proceeds to the *Decision* stage. Load miss and remote load requests are prepared in this stage, or a hit is diagnosed. In the *Access* stage, misses and remote loads only access the job-list and hits access the data arrays. For load hits an *L1 Fill* stage follows.

Other than remote stores, operations in the second category may also result in a hit during the *L1* stage, but, because of the write-through policy of the L1, an L2 allocator request is also issued. Remote stores, are identified during this cycle by the processor ART and are written to the write combining buffer in a subsequent *Store Buffer* stage. On a combining buffer hit (i.e. access that finds in the buffer data for the same destination line) processing of the remote store completes in this cycle, unless the combining buffer becomes full. In the latter case a request is issued to the L2 allocator in this stage[11]. An allocator request is also issued on a combining buffer miss (i.e. access that finds in the buffer data for a different destination or line). Loads that miss in the L1 cache, issue a request to the L2 cache allocator as well. In all cases the pipeline is stalled unless the allocator grants the request immediately.

After a grant from the allocator, stores proceed to the *Decision* stage. If the single miss status holding register (MSHR) is full and a cacheable store miss is detected the processor pipeline is stalled, but without blocking the *Decision* stage. Otherwise, cacheable store misses and remote store requests prepare a communication job description, or an L2 store hit is identified. In the *Access* stage, cacheable store misses and remote stores enqueue a job description in the job-list, while local scratchpad stores and cacheable store hits access the data arrays. In addition, store misses write their data in the MSHR.

The third category, referring to ESL accesses, is different in two ways. First, ESLs are not L1 cacheable –loads only acknowledge the L1 and return data directly

[11]The store combining buffer has a configurable timeout that allows it to also initiate the L2 allocator request independently of a processor store.

to the processor. Thus, both ESL loads and stores always result in a request to the L2 allocator during the *L1* stage. Second, local ESL accesses may trigger an event response in the *Decision* stage. This is utilized for stores to counter synchronization primitives, which may dispatch notifications if the counter becomes zero. For communication initiation, a corresponding description is enqueued in the job-list in the *Access* stage. In addition, operation side effects can be implemented for these accesses. For example, as will be discussed in the next subsection, counter loads return the value of the counter which is actually stored in the cache tags, and stores to a specific offset in a single-reader queue ESL update the head pointer actually store in the tags. Tags read and write, in *Decision* and *Access* stages respectively, are utilized to implement this kind of operation side effects.

3.2.5 Communication and Synchronization Primitive Implementation

There are three types of event sensitive lines, as discussed in subsection 3.1.2: command buffers, counters, and queues. These ESLs condense all the necessary NI state for the corresponding communication and synchronization primitives. This subsection presents the format used to store the required information and the integration of the corresponding mechanisms in the NI pipeline exploiting the event response technique. Operation metadata are stored in the ESL tag, and the data block is utilized in the cases a software interface is needed for configuration or interaction with the ESL. For counter and queue synchronization primitives, word offset zero of the ESL data block is used as the “official” address providing the corresponding access semantics discussed in chapter 2 (subsection 2.4.2).

3.2.5.1 Command Buffers

The command buffer communication primitive allows software to pass transfer descriptions to the NI, with a series of store operations. The order of the stores is immaterial. Figure 3.10 shows how the tag and data block of a command buffer ESL are formated. The tag includes bit fields for a completion bitmask, a read and a write access rights mask, the operation code, and a bit-flag indicating a remote source address useful in RDMA operations. The completion bitmask provides a bit-flag for each word in the command buffer data block. When a word in the data block is written the corresponding flag is set. Read and write access rights mask fields are intended for recording if the currently running thread has the appropriate access rights, for the use of transfer address arguments the thread requests with the

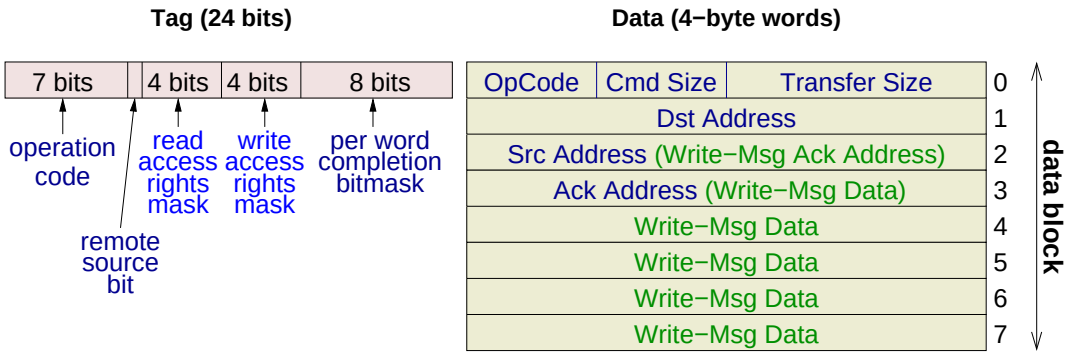


Figure 3.10: Command buffer ESL tag and data-block formats. The data-block format corresponds to software transfer descriptions for RDMA and message operation initiation.

transfer description. Their operation is not currently implemented because the address region table (ART) does not support protection yet. The operation code is used to designate the use of ESL data block words (differentiated for write messages).

RDMA-copy operations and read-messages use only the first four words of the ESL for a transfer description. In these operations, the first word of the command buffer data block (at offset 0) expects the transfer size in the two least significant bytes, the command's size (i.e. 4 words) in the next more significant byte, and the transfer command operation code in the most significant byte. When this word is written, the NI updates the ESL tag by complementing the completion mask according to the command size, writing the operation code field, and setting the unnecessary permissions in read and write access rights mask. The second word of the command buffer data block (at offset 1) expects the destination address for the transfer, the third (at offset 2) the source address, and the fourth (at offset 3) the acknowledgement address. When all words are stored in the command buffer, a transfer is initiated by enqueueing a job description in the job-list, pointing to the command buffer.

Write messages do not require a source address, so the third word of the command buffer is used for the acknowledgement address, leaving words from offset 3 to offset 7 of the command buffer data block available for message data. In addition, write messages do not use the transfer size field of the first data block word, since the command size field suffices. All stores to a command buffer read metadata during the *Decision* stage of the memory access pipeline, described in the previous section, and update them in the *Access* stage. In addition, a job description can be enqueued in the job-list during the *Access* stage, when completion of the transfer description is detected in the *Decision* stage. As a result, RDMA and minimum size write message operation initiation requires only four processor stores.

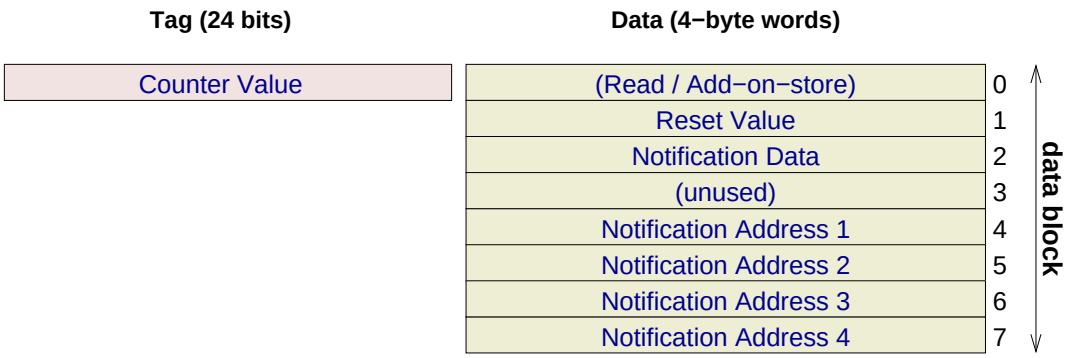


Figure 3.11: Counter ESL tag and data-block formats. The tag stores the counter value, updated atomically by the NI. The data-block is used for software configuration of notifications and counter operation. Counter read access, and atomic add of stored values is provided at word offset zero of the data-block.

3.2.5.2 Counters

Counters provide the ability of pre-configured, automated, communication initiation in the form of notifications. In addition, store accesses to the "official" address of the counter result in atomic increment by the amount of the stored value, which is treated as two's complement encoded singed number. The formats of a counter ESL tag and data-block are depicted in figure 3.11. The tag stores only the 24-bit counter value. The word at offset 1 of the counter ESL data-block, provides a reset value used to initialize the counter when it becomes zero after a store at offset 0 (described below). The word at offset 2 stores a single word of notification data, and offsets 4-7 store the addresses to which notifications are sent.

Indirect software access to the counter value is provided via offset 0 of the counter ESL data-block. Processor loads at this offset read the counter value from the tags during the *Decision* stage, which is sent back through a dedicated path[12]. Local or remote stores at word offset 0 read the counter also in the *Decision* stage, perform the add operation, and check for a zero result. The counter is updated in the subsequent *Access* stage. In order to initiate the notifications, in case the counter value becomes zero after a store, the counter ESL data-block is also read during this stage. Four notification jobs can be enqueued in the job list in the four subsequent cycles, stalling the L2 pipeline. Notification address and data are placed in the job list, and the counter is updated again with the reset value.

[12]Tags are also mapped in the processor address space to support ESL configuration, and thus support a path for load operations. This path is not shown in figure 3.8.

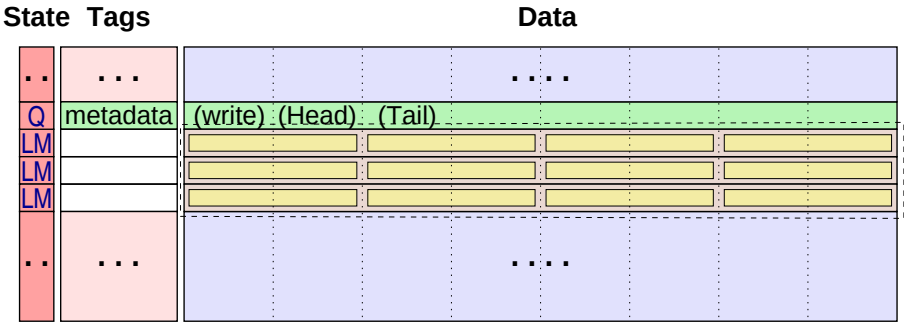


Figure 3.12: Formation of a single-reader queue in scratchpad memory of the SARC cache-integrated NI. Double-word queue item granularity is depicted.

3.2.5.3 Queues

The SARC prototype implements single- and multiple-reader queues. The support required is similar and so is the tag format. Single-reader queues (sr-Qs for short) provide in addition software access to the head and tail queue pointers. In addition to single-reader queue functionality presented in subsection 2.4.1, sr-Qs support different item granularities that are submultiples of cache-line size (i.e. 1, 2, 4, and 8 words), under software configuration. An example with double-word item granularity is illustrated in figure 3.12. Multi-word item enqueue operations must use a write message. The queue is formed in normal scratchpad memory lines consecutive to the queue descriptor ESL. As shown in the figure, the "official" word offset of the single-reader queue ESL data-block allows writing to different queue offsets (enqueue operations). Word offset 1 allows indirect loads and stores to the queue head pointer and offset 2 allows load access to the tail pointer –stores are ignored so that the tail is only modified by the NI.

In order to support queue pointers in the limited tag space, some restrictions are imposed to queues: (i) the number of scratchpad lines forming the queue body plus 1 (the ESL) must be a power of two, (ii) the whole sequence of scratchpad lines including the ESL must be aligned to its natural boundaries (i.e. the ESL must start at an address that is a multiple of the total bytes of scratchpad and ESL data storage for the queue). Thus, queues can be 2, 4, 8, etc cache lines (including the ESL), which with the 32-byte line size of the prototype corresponds to 64, 128, 256 bytes respectively; the queue descriptor ESL must be allocated to a multiple of that size.

Figure 3.13 shows the tags of single- and multi-reader queues which are similar. The single-reader queue tag includes a head and a tail pointer, an item size field, an extra bit to indicate a maximum size queue (i.e. 128 items) and a full bit. For queues of smaller size, the size is included in all queue pointers as the most signif-

icant bit of value 1. Because queue size is a power of two, its binary representation has a single 1. This 1 is of higher significance than the bits used for pointers in the range 0-queue_size. For example, a queue of maximum size of 16 words with one word item size uses head and tail pointer values of 0 to 15 (0001111 in binary) so the bit indicating the queue size of 16 (0010000 in binary) can be overlapped with each pointer without changing any of the possible values, as long as the bit indicating the size and other more significant bits are ignored (this is trivial to implement in hardware). The case of a maximum size queue that uses all pointer bits requires an additional bit, and the extra bit is used for this purpose. Bits of lower significance than the bit indicating the queue size are the *useful* pointer bits.

The multiple-reader queue tag has almost the same fields, but it uses two tail pointers, one for reads and one for writes, and two full bits. In addition, each item consumes one cache-line in order to reserve sufficient storage for read request buffering in each queue entry. This is guaranteed by dropping any data of writes over 7 words; the intended use is via messages that support up to 5 words of data.

Buffering write packets arriving from the NoC in a queue (and read packets in the case of multiple-reader queues) exploits the usual L2 access flow: in *Decision* stage, after the tags are read, some least significant bits of the accessed address (the "official" queue offset) are replaced with the useful bits of a tail or head pointer. In addition, during the *Decision* stage a possible match of a read and a write is detected for multiple-reader queues. The data arrays are written in the *Access* stage, and, if a match was detected, a job description in enqueued in the job-list.

3.2.6 Hardware Cost and Design Optimization

The hardware cost of our design was studied in [99]. The results of this study show that the the outgoing control and datapath of the integrated NI is about twice that of a simple cache, or $2/3$ of the total cost for a simple cache. As the prototype matured and in light of the hardware cost measurements of [99], it became apparent

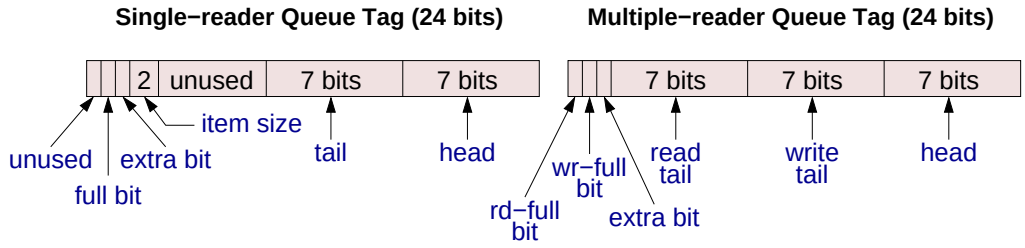


Figure 3.13: Single- and multiple-reader queue tag formats.

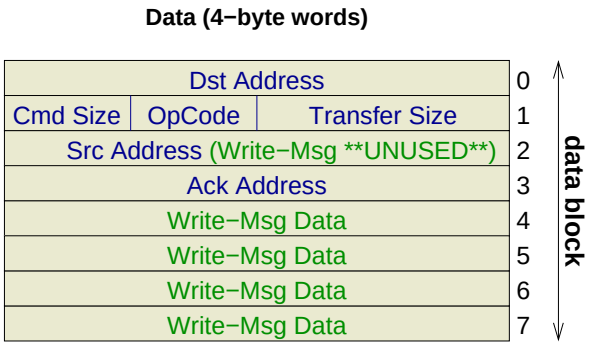
that there was room for improvement of the NI design. We studied the optimization of the prototype's hardware cost to improve cache-integration quality and reduce the design's footprint.

We based the optimization approach on the fact that RDMA and message descriptions formed in command buffers, provide most of the information that appears in NoC packet headers. This information also appears, for the most part, inside multiple-reader queues as response and acknowledgement addresses of read requests. We modified the format of the command buffer data-block, used to describe transfers, and the NoC packet formats, so that they favor each other, as shown in figure 3.14. The optimized formats can be compared against the original formats in figures 3.7 and 3.10. The modifications respect alignment of fields in correspondence, to avoid redundant multiplexing for a 64-bit wide NoC.

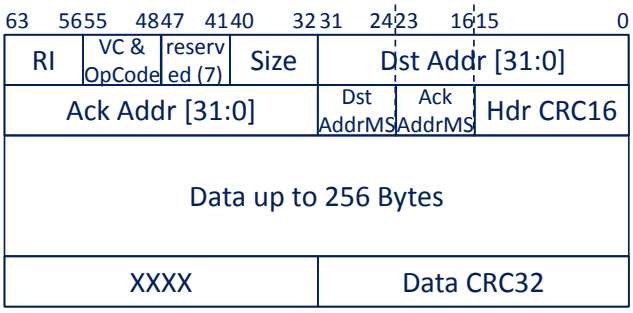
The result was the reduction of the NI-out logic to 1206 LUTs and 513 flip-flops (33.2% reduction) and also the reduction of the NI-in logic to 834 LUTs and 458 flip-flops (23.4% reduction), with only a minor increase in the memory controller and its datapath. Assuming that 1 LUT or 1 flip-flop is equivalent to 8 gates, the hardware cost of NI-out was reduced to 13.7K gates (from 20.6K), and that of NI-in to 10.3K gates (from 13.5K).

Figure 3.15 shows the results we got from this optimization, as a percentage of the gate count for a plain cache. Bars of the unoptimized design from [99] are also shown on the right side of the chart, separated from other bars with a vertical line, for comparison. Bars on the left of the vertical line correspond to optimized designs. The simple integrated design (third bar from the left) is less than 20% larger than the plain cache. The results of [99] provide a comparison with a partitioned cache and scratchpad design with simple RDMA support to a corresponding simple integrated design. In that study, the integrated design (second bar from the right) is 35% less than the partitioned (first bar from the right), where after optimization the simple integrated design is 42.2% smaller than the corresponding partitioned design (fourth bar from the left). This increase is in spite of reduction of the partitioned design to 63.4K gates after optimization, versus 72K gates before optimization.

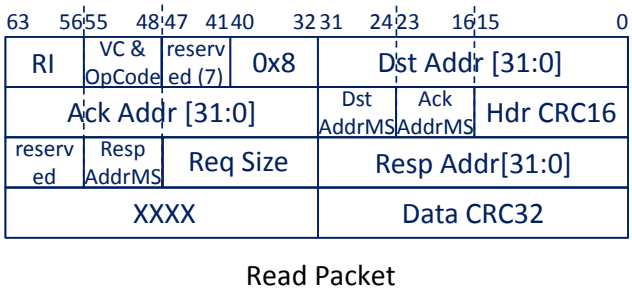
In addition, figure 3.15 presents the cost of adding more advanced functionality to the simple integrated design in bars for "Integrated advanced" and "Full functionality" designs. The addition of counters and notifications requires logic equivalent to 13.6% of the plain cache design, or 4.2K gates. Adding then multiple-reader queues requires 15.2% or 4.7K gates, single-reader queues 10.6% or 3.3K gates, barrel shifter for RDMA arbitrary alignment 9.6% or 3K gates, and combining buffer for remote stores 23.8% or 7.3K gates.



(a) Modified command buffer data-block format.



(b) Modified NoC write packet format.



(c) Modified NoC read packet format.

Figure 3.14: Modified command buffer and NoC packet formats for design optimization.

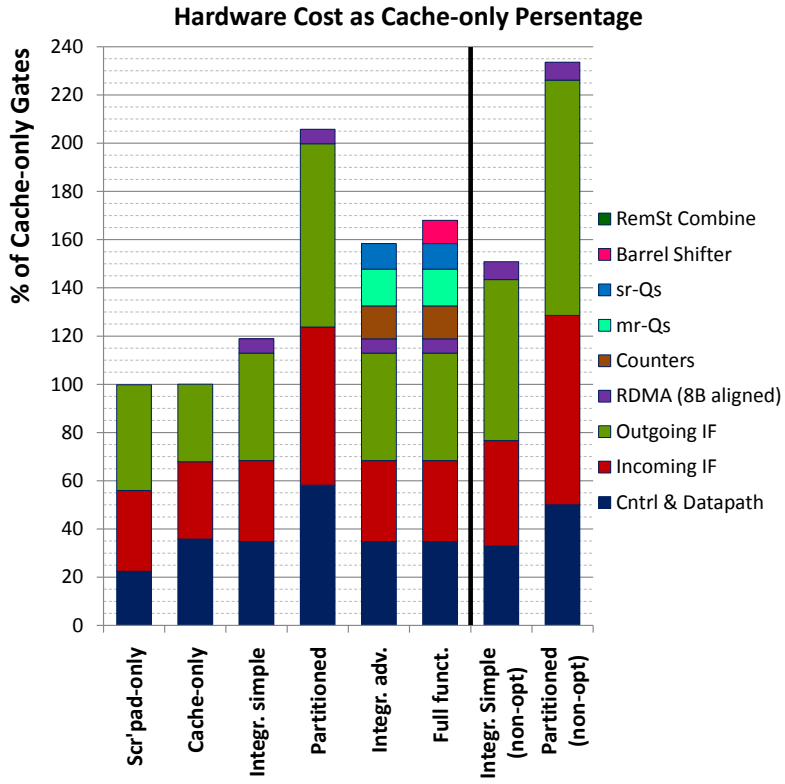


Figure 3.15: Area optimization as a percentage of a cache-only design.

Putting these results into perspective for cache integration in future processors, one should keep in mind that the base cache-only design of figure 3.15 only supports a single outstanding miss and miss status holding register (MSHR). In addition, these results only include the control and datapath of the cache and the integrated NI designs, and thus do not reflect the area cost of the SRAM arrays for tags and data of the cache and scratchpad, that will occupy most of the area in the integrated design.

Nevertheless, changing the packet format requires changes in all system modules of the prototype (NoC, DDR controller and its DMA engine, L2 NI and off-chip switch) and associated debugging for robustness. In addition, changes in software library code are required because of the modification of transfer descriptions using the old command buffer data-block format. Because of these reasons and because only 65% of the FPGA logic resources are utilized with the non-optimized design, the SARC prototype sticks, at the moment, to the stable but unoptimized version.

Collaborators: A number of people contributed to the design and implementation efforts of the hardware prototype implementing the SARC network interface. George Kalokerinos and George Nikiforos implemented most of the design. Xiaojun Yang implemented the DRAM controller and the off-chip network interfaces, and Christoforos Kachris implemented the crossbar NoC. George Nikiforos was instrumental in the design optimization of subsection 3.2.6. Michael Zampetakis and Pranav Tendulkar served as software developers for the prototyped platform, helping to locate and debug several corner cases of cache and NI interaction. Vassilis Papaefstathiou was contributory to the design and implementation of many aspects of the cache-integrated NI and provided, in collaboration with George Nikiforos, the measurements for the non-optimized design of subsection 3.2.6.

4

Evaluation

Evaluation of a new architecture is a difficult task. Performance of software run on new hardware is not directly comparable with performance on existing platforms, which have different architecture and clock rates. Furthermore, a hardware prototype has limited resources, which makes comparison with other systems infeasible. In addition to these issues, coherence-based implicit communication support on the SARC prototype is still under development, so comparison with explicit communication or exploitation of their coexistence is not possible yet. Simulation can be used to overcome these issues. In the case of the SARC architecture, the main obstacle in such simulation-based evaluation, was the scarcity of software that could exploit the on-chip communication efficiency of chip multiprocessors, and, to some extend, the lack of multiprocessor software expertise in the group.

For a long time, software targeted uniprocessor systems, so communication was not an issue at all (at least for software). The advent of CMPs found the software community largely reluctant to adopt parallel algorithms as a reasonable effort approach to the use of multiple cores. What is more, future workloads that would be representative for the general purpose domain, although less of an issue for uniprocessors, puzzled academia and industry for some time in view of future CMP hardware. Both the amount of "CMP-friendly", throughput-oriented workloads, and the potential cost of writing and debugging parallel software contributed to this quest.

Moreover, most software for past multiprocessors targeted either implicit or explicit communication, using largely different algorithms for the same application. In the former case, inter-processor communication (IPC) was only indirectly addressed, but usually with specific data partitioning to processors and parallel algorithms. In the case of software platforms for hardware supporting explicit communication (e.g. MPI), intra-node memory space was managed at very coarse granularities and by multiple software layers. These past approaches do not account for the limited on chip space. The importance of this issue, at least for the explicit communication paradigm, can be viewed in the difficulty faced in writing efficient software for the Cell processor, which can be reason for scientific publication in some forums. In any case, software for one of the models or both should be written from scratch.

The following sections provide a partial evaluation of the SARC architecture and cache-integrated network interfaces. Section 4.1 presents results for communication mechanism performance on the hardware prototype. Section 4.2 provides simulation results on the use of cache-integrated network interface synchronization support for locks and barriers using microbenchmarks. Finally, section 4.3 presents simulation-based evaluation of the use of multiple-reader queues for task scheduling augmenting the Cilk run-time system.

4.1 Evaluation on the Hardware Prototype

In this section, microbenchmarks are used to evaluate the performance of locks and barriers with the four processors available in our FPGA prototype. In addition, STREAM, FFT, and Bitonic sort benchmarks are used to evaluate NoC bandwidth utilization, and the effectiveness of direct communication using RDMA and remote stores. Benchmark data sets are kept small enough to fit in scratchpad memories. The objective of the evaluation is to demonstrate that the proposed architecture achieves low core-to-core communication latency, low latency for high-level synchronization primitives (i.e. locks and barriers), effective utilization of the available on-chip and off-chip memory bandwidth and exploitation of fine-grain parallelism in applications.

4.1.1 Software Platform and Benchmarks

Software was compiled with a version of gcc (mb-gcc), targeted to the Microblaze processors. Xilinx Embedded Development Kit (EDK) allowed code mapping on "bare metal", and the Xilinx Microprocessor Debug (XMD) engine was employed for run-time debugging. The UART module of the prototype (see figure 3.6) was also utilized for a single I/O terminal.

An SHMEM-like library was used to circumvent the need for a special compiler targeting a global address space, and for a method to exchange addresses among threads in absence of cache coherence[1]. SHMEM [100] provides a hybrid shared memory/message passing programming model. It uses explicit communication and management of memory and object replication is in software, which is similar to message passing, but does not use two-sided send-receive style operations. Instead, it is based on one-sided, non-blocking operations that map directly to the explicit communication mechanisms provided in the SARC architecture. Remote memory address specification in SHMEM would use a thread identifier and a local address. Such pairs were mapped to global addresses with dedicated library calls for our platform.

Three libraries were implemented, *syslib*, *scrlib*, and *nilib*. The *syslib* implements system management functions for hardware components other than the cache-integrated NI. It provides a lock implementation using the hardware lock-box

[1] Alternatively, exchanging addresses among threads could be achieved maintaining cache coherence in software and per-thread signaling scratchpad areas, or via non-cacheable, off-chip, shared memory.

module accessible over the platform's shared OPB bus. It also provides a centralized barrier implementation which also uses the lock-box for locking. In addition, *sylib* supplies thread-safe main memory allocation and I/O facilities, and access to hardware registers for thread identifiers and cycle accurate global time.

The *scrlib* library allows manipulation of NI memory, to allocate parts of L2 cache as scratchpad regions at runtime, and designate the use of cache lines for communication and synchronization primitives (command buffers, counters and queues). It also provides functions to convert local addresses to remote ones, and check if an address is local or remote. Last, *nilib* contains functions for preparing and issuing DMAs and messages via command buffers, managing counters and notifications, and accessing single-reader queues.

The lock microbenchmark measures the average time for a lock-protected, empty critical section, when 1, 2, or 4 cores contend for the lock. Lock acquisitions start after a barrier, and 10^6 repetitions per processor are measured (the time of loop overhead code is subtracted). Two lock implementations are compared: the one in the *syslib* using the shared hardware lock-box, and an implementation using a multiple-reader queue with a lock-token as presented in chapter 2 (see figure 2.14(a)). The barriers benchmark similarly measures the average time for a barrier in 10^6 invocations, after an initial barrier. The *syslib* centralized barrier[2] is compared to a barrier implementation using a single counter synchronization primitive and notifications.

The STREAM triad benchmark [101] is designed to stress bandwidth at different layers of the memory hierarchy. The benchmark copies three arrays from a remote to a local memory, conducts a simple calculation on the array elements and sends the results back to original remote memory. Two configurations of STREAM were developed for stressing on-chip and off-chip memory bandwidth respectively. In the on-chip configuration, the data are streamed from scratchpad memories to scratchpad memories and backwards, whereas in the off-chip configuration, data are streamed between DRAM and scratchpad memories. In each configuration, multi-buffering was applied to overlap the latency of computation with communication from and to "remote" memory, varying the number of buffers and their size. In addition, different versions of the benchmark utilize RDMA and remote stores for producer-initiated transfers (transfers *from* remote memory are always consumer-initiated to preserve benchmark semantics).

The FFT benchmark originates from the coarse-grained reference FFT implementation in the StreamIt language benchmarks [102] and does a number of all-to-

[2]The *syslib* implementation integrates the counter of processors arriving at the barrier and a global sense in a variable protected by the lock-box.

(a) Contended lock times

	Lock-box			**mr-Q Lock**		
CPUs	1	2	4	1	2	4
Acquire	44	87	201	42	82	193
Release	32	32	32	21	21	21
Total	76	119	223	63	103	214

(b) Barrier times

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

Table 4.1: Different lock and barrier latencies.

all data exchanges between the processors. RDMA and remote stores alternatives are explored for the data transfers, to explore tradeoffs between the two communication mechanisms. Evaluation varies the input data-set size and the core numbers used. The bitonic sort benchmark also originates from the reference implementation of the StreamIt language benchmarks [102]. Bitonic-sort has a low communication to computation ratio and we use it to measure the minimum granularity of exploitable parallelism on the prototype. The benchmark measurements exclude the initial fetching of data in scratchpad memories. The trade-off between RDMA and remote stores is also evaluated in this case, varying the data-set size.

4.1.2 Results

Table 4.1(a) shows acquire, resease and average latencies for contended lock-unlock operations using the lock-box or a multiple-reader queue (mr-Q). The lock that leverages the multiple-reader queue executes an acquire-release pair for an empty critical section under full contention between 4 cores in 214 cycles. Although the lock-box provides comparable performance for the small number of contending processors, multiple-reader queue access via the crossbar, including controller times is faster regardless of the number of processors. Table 4.1(b) shows centralized and counter-based barrier times with one, two, and four participating processors. The counter-based barrier uses on-chip signaling for barrier arrival and automated notifications between 4 cores takes 117 cycles, compared to 618 for a centralized barrier over the shared OPB bus of the prototype.

To put these numbers in perspective, consider that the one-way latency of a

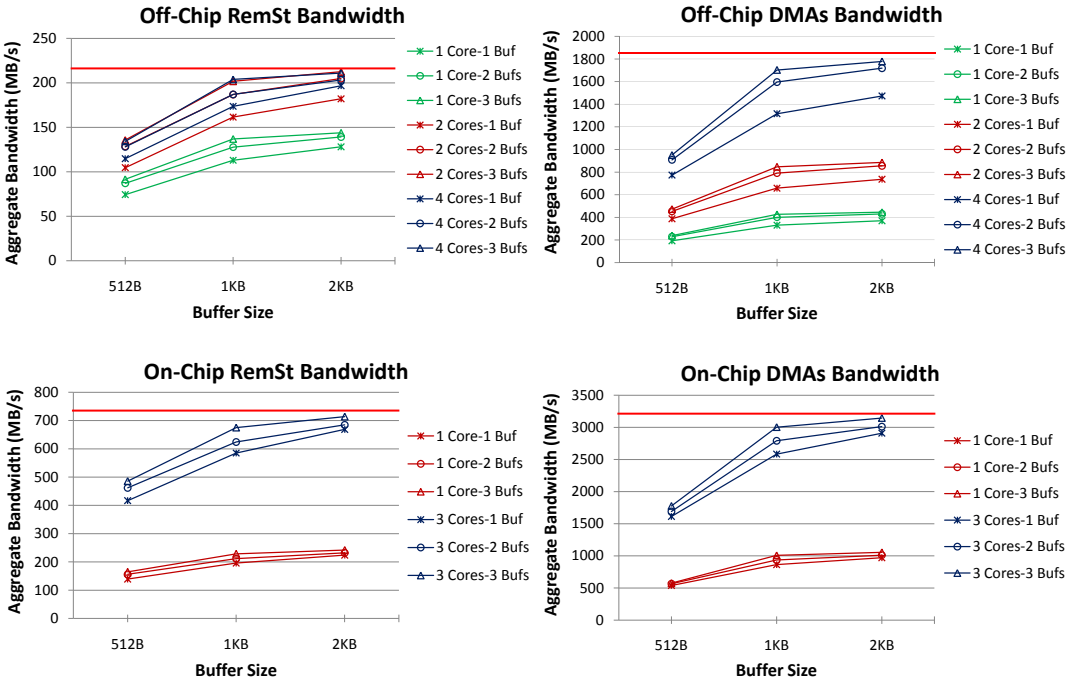


Figure 4.1: Performance of STREAM triad benchmark.

remote store (used for signaling) is 18 cycles, and of a read message (used for lock acquisition) is 21 cycles in hardware simulation (for timing breakdown of hardware communication mechanisms see [99]). Although direct comparisons with other hardware designs are not possible because of differences in CPI and the limited size of resources in the prototype, note that on leading commercial multicore processors, lock acquire-release pairs cost in the order of thousands of cycles, and barriers cost in the order of tens of thousands of cycles [103]. Considering processors with multi-GHz clocks, lock-unlock time corresponds to hundreds of nanoseconds and barrier corresponds to a few microseconds[3]. These times reflect library and OS overhead that are difficult to avoid without hardware support.

Figure 4.1 illustrates the results of the STREAM benchmark on the FPGA prototype, as the achieved bandwidth varying the size of the transfered buffer. The maximum feasible bandwidth with each communication mechanism is plotted with a horizontal line. Bandwidth for remote stores and RDMA is shown in separate charts with groups of three curves colored in red, green, and blue, that correspond to benchmark runs with different numbers of cores. The three curves of each group

[3] Because the SARC prototype runs only at 75 MHz, the wall clock time for acquire-release of an mr-Q based lock is 2.782 us, and for counter barriers 1.521 us.

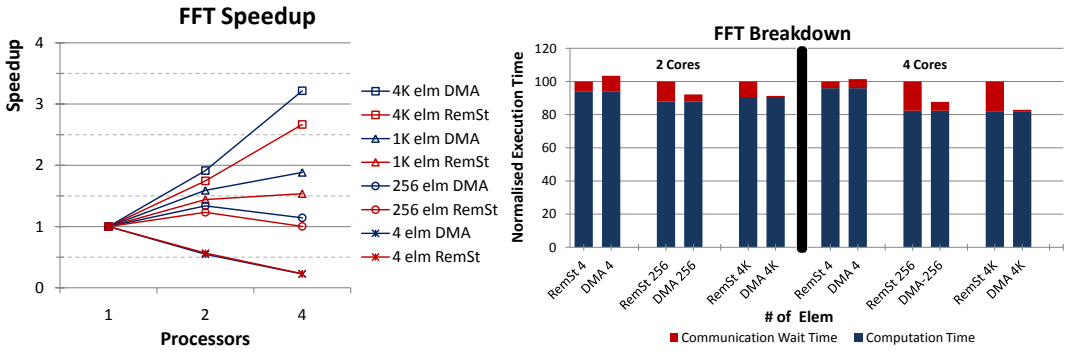


Figure 4.2: Performance of FFT.

represent use of one, two, and three buffers for computation and overlapping transfers.

Off-chip bandwidth is depicted in the upper charts, for remote stores and RDMA, measured by streaming data from three large arrays (more than scratchpads can fit) in DRAM. For measuring the on-chip realizable bandwidth data are laid out in scratchpad memories of one or two cores and the remaining cores stream data from these scratchpads. As expected, the achievable maximum bandwidth with remote stores is lower (about $7\times$ - $8\times$) than the maximum achievable bandwidth with RDMA, despite use of the store combining buffer, because remote stores incur the overhead of one instruction per word transferred whereas RDMA can transfer up to 64 KB (L2 size) worth of data, with overhead of only 4 instructions. In all cases, the architecture can maximize bandwidth and overlap memory latency using small scratchpad buffer space (3KB - 4KB), when four cores are used.

For the FFT and bitonic sort benchmarks, producer-initiated transfers are used and the time the consumers wait for data is measured. In the remainder of this evaluation, consumer wait time is referred to as *communication time*, and the remaining execution time, including the time to initiate communication, is referred as *computation time*. Note that *communication time* does not reflect the actual time for communication, but rather the amount of communication time that benchmark execution could not hide, which is the actual communication overhead for the benchmark. The left side of figure 4.2 plots the speedup of on-chip FFT for various input sizes, using remote stores and RDMA. The right side of the figure illustrates remote store and RDMA benchmark version execution time, broken down in *computation and communication times*, for selected input sizes and normalized to the time of the corresponding remote store execution.

The results exhibit a trade-off between communication with RDMA and com-

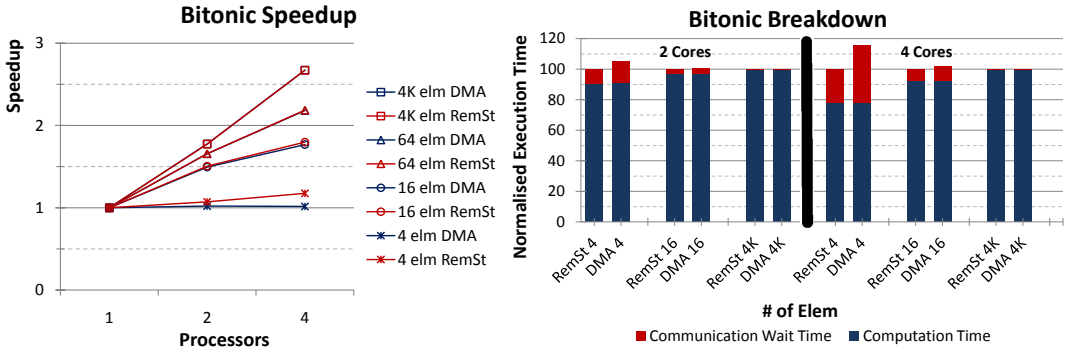


Figure 4.3: Performance of bitonic sort.

munication with remote stores. For input sizes corresponding to 8 scratchpad lines of data or less (N≤128 elements) remote stores reduce *communication time* by 5-48% and increase overall performance by 0.4-4.5%. However, FFT does not profit from parallelization on small input sizes (N≤128), because execution time is dominated by overhead for locating the receivers of messages during the global data exchange phase and loop control code that is inefficient on the Microblaze processor. For larger input sizes, the RDMA benchmark version outperforms the remote store one (by 0.6-20% in terms of overall performance) due to lower communication initiation overhead and better overhead amortization. The performance advantage of RDMA is consistently amplified as the input size increases. Parallel efficiency with RDMA reaches 95% on 2 cores and 81% on 4 cores, while with remote stores it is capped at 88% on 2 cores and 68% on 4 cores, for data-sets that fit on-chip.

Figure 4.3 shows the corresponding data for bitonic sort, which also expose the trade-off between RDMA and remote stores. In addition, the speedup chart shows that bitonic short execution provides a speedup of more than 1 even with a data-set of N=4 elements. This reflects the profitable parallelization of tasks as small as 470 clock cycles, achieved by cache-integration of explicit communication mechanisms.

In input sizes corresponding to 4 scratchpad lines of data or less (N≤64 elements), *communication time* with remote stores is 5-41% less than *communication time* with RDMA. With the same small input sizes, overall performance with remote stores exceeds performance with RDMA by 0.2-14%. For larger input sizes, *communication time* with DMAs is 13-32% less than *communication time* with remote stores, however overall performance with DMAs exceeds only marginally performance with remote stores (by no more than 0.2%) due to the low communication to computation ration of the benchmark. Parallel efficiency with RDMAs reaches 89% on 2 cores and 67% on 4 cores, while with remote stores it is slightly lower (87%

on 2 cores, 61% on 4 cores), for data sets that fit on-chip, i.e. do not exceed the 64 KB of available scratchpad space. Overall, the presence of both communication mechanisms enables more effective parallelization depending on task input size.

4.1.3 Summary

In order to exploit many-core processors with scalable on-chip networks, scalable communication mechanisms will be required, that can effectively hide remote memory access latency, and utilize available memory bandwidth. Block transfer facilities from and to local memories, like RDMA provided in the SARC architecture, achieve these targets by means of fast transfer initiation with only four store instructions, and by amortizing NoC header and CRC overhead over larger packets (4 scratchpad lines in the SARC prototype). Running the STREAM benchmark, transfers summing up to 3KB-4KB total size from 4 cores, suffice to saturate the prototype's bandwidth.

Locality will be very important on many-core processors, and effective exploitation will benefit from the use of low latency mechanisms to provide fine-grain parallelization. On the SARC prototype, remote store communication allows profitable parallelization of bitonic sort tasks of less than 500 clock cycles length. Tailoring communication mechanisms for low latency and for high bandwidth transfers can enable software trade-offs in exploiting locality when possible, versus high bandwidth overlapped transfers that naturally hide latency, when locality is not a choice.

The use of network interface synchronization primitives for lock and barrier operations provides promising performance on the prototype. The limited number of cores makes the evaluation inconclusive, so these functions are further investigated in the next section.

4.2 Evaluation of Lock and Barrier Support of the SARC Network Interface

4.2.1 Microbenchmarks and Qualitative Comparison

In order to assess the effectiveness of the synchronization primitives in the SARC architecture, we implemented barriers and locks using counters and queues. For a hierarchical barrier, counters are organized in an arrival and a broadcast tree, as shown in chapter 2 (subsection 2.4.3). The same subsection also describes how a

lock service is implemented using a multiple-reader queue. Counter-based barriers and multiple-reader queue based locks are compared, with simulation of microbenchmarks, against tree-barriers and MCS-locks from [2] that use load, store, and atomic[4] operations on cacheable variables. The barrier microbenchmark measures the average times of 10^{10} barrier episodes, after an initial barrier. Similarly, the lock microbenchmark measures the average time for 10^{10} empty critical sections (lock-unlock operation pairs) on each participating processor, from which the average across processors is calculated.

Figure 4.4 shows C-like pseudocode of the algorithm for the tree barrier published in [2], adopted to a cache-coherent memory system. Threads are organized in a tree with four children per node. On the left side of the figure, the *nodeInfo* struct shows the barrier data related to each node of the tree. There are *childready* flags updated from children nodes, *havenochild* flags keeping the values that *childready* flags must have at the beginning of barrier episodes, a *parentflag* pointing to one of the *childready* flags in the node's parent, a *rootflag* updated by the root node when the barrier is complete, and a *sense* value per node, to separate successive barrier episodes. In the original algorithm, there was a second, broadcast tree also, which is optimized for a coherent system using a single *rootflag* pointer. The broadcast tree variant was also simulated with worse results –this was predicted in a footnote of the original publication. In addition, *childready* flags are placed in separate cache lines, because our blocking directory and MOESI protocol cannot overlap multiple updates in the same line, increasing the tree-barrier time.

When a child enters the barrier, it waits "signals" from its children in the *childready* flags, and then notifies its parent by writing to the *parentflag* pointer. Then, if the node is not the root-node, it waits until the *rootflag* is updated with the episode's value of *sense*. When the root-node is signaled by all of its children, it updates the *rootflag* variable. All nodes reverse their *sense* before exiting a barrier episode.

The figure also shows where misses occur in each barrier invocation. Before the while of line 7 becomes false, four misses will occur in lines 5 and 6, one for each *childready* flag update. The update via the *parentflag* pointer at line 10 will cause a miss because the parent node had accessed it in the previous barrier episode to read its *childready* values in lines 5 and 6 –the root-node is an exception to this because its *parentflag* pointer is dummy. All nodes except the root-node will exhibit a miss in line 12 when the parent updates the variable pointed by *rootflag*. Finally, the root-node will also exhibit a miss when updating that variable in line 14, because all nodes have read the value in the previous barrier episode (exhibiting the miss of

[4]We use the SPARC v9 ISA in our simulations, which provides compare-and-swap as well as swap atomic operations required for the MCS lock implementation.

```
typedef struct treeNode {
  INT childready[4];
  INT havenochild[4];
  int* parentflag;
  int* rootflag;
  int sense;
} nodeInfo ;

typedef nodeInfo* Barrier;

(INT ==> one int per cache-line)

(child's parentflag points to
 parent's childready[x])
```

```
1  tree_barrier( Barrier B ) {
2    nodeInfo* I = &B[myID];
3    INT* childready = B[myID].childready;
4    do {                          miss x 4
5      flag = childready[0] + childready[1] +
6             childready[2] + childready[3];
7    } while( flag != 4 );
8    for(int i = 0; i < 4; i ++)
9      childready[i] = havenochild[i];
10   *(I->parentflag) = 1;miss
11   if (myID != 0)      miss
12     while( *(I->rootflag) != I->sense );
13   } else {
14     *(I->rootflag) = I->sense;miss
15   }
16   I->sense = 1 - I->sense;
17 }
```

Figure 4.4: Tree-barrier pseudocode from [2] using cacheable variables.

line 12).

In total, all nodes will suffer six misses (the root-node only 5) in the process of the three barrier steps: (i) receive signals from children (4 misses), (ii) signal my parent (1 miss), and (iii) receive –or send– the episode end-signal (1 miss). The latency of misses for *childready* flags at a parent node, is partially overlapped with the latency of misses for updates via the *parentflag* pointer at the children. Similarly, the latency of non-root node misses for receiving the episode end-signal, is partially overlapped with the latency of the root-node miss for sending the signal. Nevertheless, the *childready* flag misses at each level of the tree are serialized by the while of line 7. In other words the latency of overlapped misses is multiplied by the number of levels in the tree minus one. Each of these misses entails a round-trip to the child or the parent updating the relevant flag via the directory. The final miss on the variable pointed by *rootflag* is also added to the total for each node, but incurring different latency in each case since the directory can only reply to one load request at a time. The requests are overlapped with its other and with the invalidations the directory sends when the root node updates the flag. With rough calculations, for T threads there are $log_4(T) + 1$ round-trip transfers through the directory, plus T serialized packet injections in the NoC.

The same three signaling steps are also utilized in the counter-based barrier with the following differences: (a) signals do not require a round-trip and do not go through the directory, but go directly to the receiver, (b) since no chain of misses is involved, a broadcast signaling tree is exploited as well, to avoid sending all the signals from a single node. The total in this case is $2 \times log_4(T)$ one-way transfers. Half of them are suffered by all nodes, and the other half incur different latency at

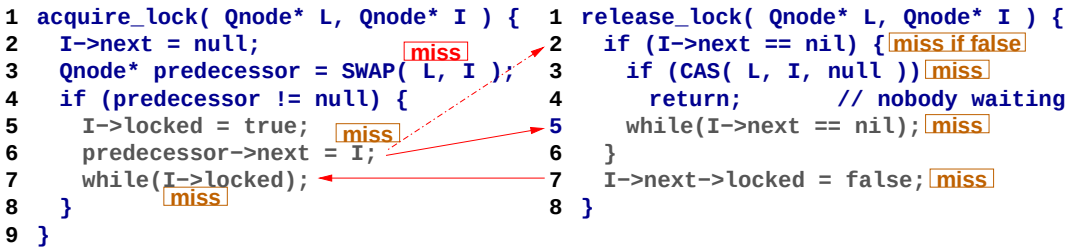


Figure 4.5: MCS lock pseudocode from [2] using cacheable variables.

each level of the tree.

Figure 4.5 shows the MCS lock algorithm from [2]. A queue of waiting threads is formed in the case of contention for the lock. Each thread is represented by a *Qnode* struct, that has a next pointer (of *Qnode** type), and a boolean flag *locked*. A cacheable shared pointer *L* points to the last *Qnode* linked in the queue. When acquiring the lock, a thread resets the next pointer in its *Qnode* and atomically sets *L* to point its *Qnode* with the swap (SWAP) operation in lines 2 and 3 of *acquire_lock*. Lock acquisition, when nobody has the lock, finds a *predecessor* value null at line 4 of *acquire_lock*. Reversely, lock release before anyone enters the wait queue, succeeds in atomically removing the thread's *Qnode* from the *L* pointer with the compare-and-swap (CAS) at line 3 of *release_lock*. No queue is formed and only the dark blue part of the code is used in these cases.

In the case of contention, the highlighted in dark green part of the code is used to link the queue in *acquire_lock*, and to wait for this linking before forwarding the lock in *release_lock*. A *predecessor* may be found in line 4 of *acquire_lock*. In this case, the requestor sets the *locked* flag in his *Qnode* to true, links the *Qnode* in the queue after that pointed by *predecessor*, and waits to be signaled by the predecessor thread (lines 5, 6, and 7 of *acquire_lock*). Reversely, if a thread releasing the lock is the predecessor of a thread being linked in the queue, it may find an updated next pointer in its *Qnode* at line 2 of *release_lock*, or the CAS may fail at line 3 because *L* points to a successor.

Because linking of a successor in the queue is not atomic, the releasing thread must wait for it to be complete, and deliver the lock. The red arrows in figure 4.5 show that either statement 6 of the thread executing *acquire_lock* will execute before statement 2 of the thread executing *release_lock*, or statement 5 in *release_lock* will wait for statement 6 in *acquire_lock*. In effect, statement 7 in *release_lock* will always execute when the next pointer of the releasing thread is updated and the successor is linked. In addition, statement 7 in *acquire_lock* will wait for statement 7 in *release_lock*.

The figure also shows where misses occur in execution of a *acquire_lock-release_lock* pair. Without contention, no queue is formed and only the node acquiring and releasing the lock accesses the *L* pointer. Only one miss may occur in line 3 of *acquire_lock*, when another thread had acquired (and released) the lock before the current. This is the red miss in the figure. When there is contention, 4 or 5 more misses will occur, shown in brown in the figure. First, updating the *next* pointer in the predecessor's *Qnode*, in line 6 of *acquire_lock*, will miss because the previous time the predecessor thread released the lock it had accessed that pointer either in line 2 or in line 5 of *release_lock*. Second, the while in line 7 of *acquire_lock* will miss after the update of the *locked* flag from the releasing thread. Reversely, when the releasing thread finds a successor in the queue, statement 2 may miss if the *next* pointer in its *Qnode* has been updated from the successor thread. If statement 2 does not miss, statement 3 will miss because the successor has accessed *L* in line 3 of *acquire_lock*, and statement 5 will miss after the *next* pointer is updated at line 6 of *acquire_lock*. Finally, statement 7 of the releasing thread will miss because the successor thread had accessed the *locked* flag the previous time it had acquired the lock at statement 7 of *acquire_lock*.

In summary, when there is contention, two "signals" are propagated between a "successor" and a "predecessor"; one for the update of the predecessor's *Qnode next* pointer, and one for the update of the successor's *Qnode locked* flag. In addition, one or two misses will occur to transfer the line hosting the shared *L* pointer to the two threads, for a total of 5 or 6 misses. When the successor thread executes statement 6 of *acquire_lock* before the predecessor initiates the release, only 5 misses occur. When the successor executes *acquire_lock* at about the same time the predecessor executes *release_lock*, it is more likely to have overlap of misses in the two threads, but it is also more likely to have 6 instead of 5 misses.

Misses for *L* of successive *acquire_lock* calls can be overlapped, and the same can be true for the misses at line 6 of *acquire_lock*. The other misses are serialized because only one thread at a time can call *release_lock*, and their cumulative overhead is suffered. With T threads, this results in about $T \times 3$ misses per critical section, after the initial T-1 roughly corresponding to the first critical section of each thread.

In comparison, accessing a lock-token in a multiple-reader queue, does all queue pointer changes in hardware, with each requestor constructing only a read message locally and exhibiting a local miss when the lock token is delivered (L1-invalidate, L2-hit). As opposed to signaling with coherent transfers, no directory indirection occurs for the transfers to and from the multiple-reader queue. Furthermore, the lock requests (reads) and the lock-token release (write) use independent

L1 caches	32KB, 64 byte blocks, 2-way, 1 cycle latency
NIs/L2 caches	256KB 64 byte blocks, 16-way, unified, core-private, exclusion of L1s, single port, 2 cycle latency remote non-coherent access, 7 cycle latency, MOESI coherence
Directories	4 (P=16, 32) or 8 (P=64, 128), P protocol engines in total, serve address space segment, state for all cached lines in segment, blocking controller, 10 cycle latency
NoC	4 virtual networks, infinite buffers, pckts: 80 byte (data), 16 byte (control) on-chip links: 1 cycle, 128 bits, switch latency: 3 cycles
Memory	130 cycles latency

Table 4.2: Memory hierarchy parameters of the simulated system.

subnetworks, allowing the release to bypass queued requests at the node hosting the multiple-reader queue. This is not the case with coherence and atomic operations, where requests of multiple SWAP operations on the *L* pointer to acquire the lock, can be queued in front of a CAS operation on the *L* pointer to release the lock, in the case of contention. Nevertheless, one round-trip of the lock-token per contending thread is suffered, totaling T round-trips plus T local message compositions plus T local misses latency per critical section, after (roughly) the first critical section of each thread.

When the time for the average critical section increases, under high contention (successive critical sections by multiple threads), time proportional to the average critical section time and the number of contenders will be added in the wait time of each thread. Reducing contention by adding an interval between successive critical sections of a thread, will probably have little effect if the interval does not exceed the average critical section time multiplied by the number of contenders. When it does, though, the MCS lock will become almost as fast as the multiple-reader queue based lock, because of incurring a single miss and directory overhead will no longer be amplified by multiple misses per acquire-release pair. If the same thread's critical section is executed multiple times before another thread requests the lock, the MCS lock should be faster incurring no misses, compared to the necessary lock-token round-trip with the multiple-reader queue.

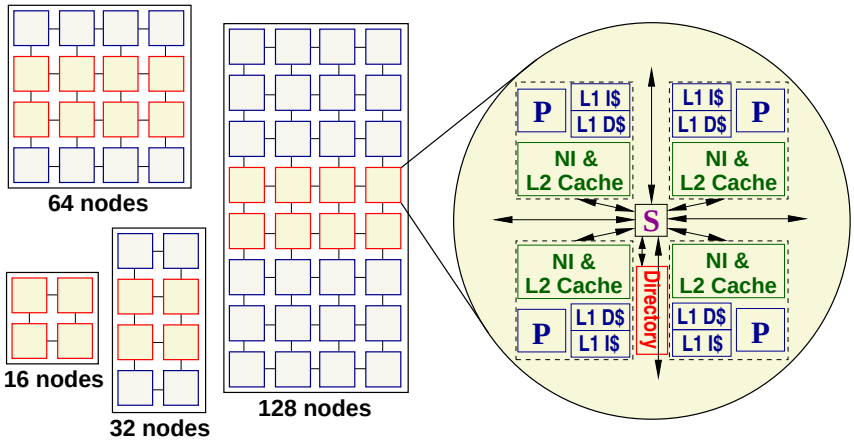


Figure 4.6: NoC topology for 16, 32, 64 and 128 core CMPs.

4.2.2 Simulation methodology

For the evaluation of *event responses* GEMS [104] is used as a timing simulator, driven by memory accesses supplied from Simics [105] full system simulation. GEMS provides a parameterized coherent memory hierarchy and includes a domain-specific language, SLICC, to formalize the definition of coherence protocol controllers. Controller events and cache line states are defined in SLICC and state transitions are associated with sets of actions. The L2-cache integrated network interface, is modeled extending the coherence controller to support direct access (without tag matching) to scratchpad regions of locked lines, and adding new states as well as state transitions for processor and network events that capture the behavior of event sensitive lines.

The configuration of the memory hierarchy for the simulated system is shown in table 4.2. To the best of the author's knowledge, there are not any proposals for directory and memory controller organization of chips with 16 to 128 processors, as those modeled. The number of memory controllers cannot scale to the number of processors in such chips. Instead of separating directories from memory controllers, which would require an additional hope for off-chip accesses, the directories are kept with memory controllers. In addition, state only for on-chip cache memory is assumed in each directory, and an optimistic latency of 10 cycles (e.g. using directory caching).

The on-chip network topology modeled is a concentrated mesh, shown in figure 4.6, that exploits relatively large valency switches, as expected in large CMPs, to reduce network hop-distance. The figure shows red and blue squares, corresponding to four nodes each, where red squares include a directory, as shown on the right side

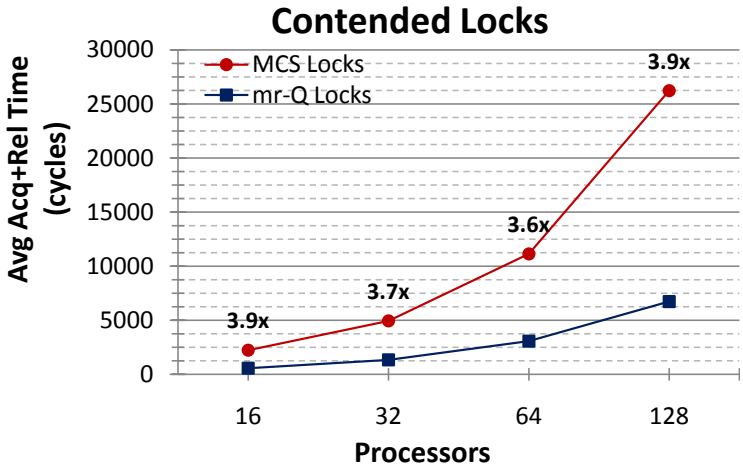


Figure 4.7: Average per processor acquire-release latency across the number of critical sections versus the number of cores, for simulated MCS locks using coherently cached variables and multiple-reader queue (mr-Q) based locks accessed over a non-coherent address space portion.

of the figure, and blue squares do not. The concentrated mesh was chosen because of the more uniform hop-distance of nodes to directory controllers placed in the middle of the topology.

The combination of GEMS and Simics behaves as an in-order, sequentially consistent system. A set of libraries, similar to *scrlib* and *nilib* of subsection 4.1.1, where developed to run over Simics. For our measurements we use the Simics support for light weight instrumentation, using simulation break instructions, to selectively measure synchronization primitive invocation intervals excluding the surrounding loop code.

4.2.3 Results

Figure 4.7 shows the average latency of contended lock-unlock pairs of operations. In both implementations requests are queued until the lock is available. The multiple-reader queue (mr-Q) based implementation is about 3.6-3.9 times faster than the MCS lock implementation. The MCS average time for lock-unlock operation pairs exceeds the expected time of $2\times$ to $3\times$ the average time for lock-unlock via the mr-Q, with the calculations of subsection 4.2.1. The additional time should be accounted to the per miss directory indirections, and the limited ability of the directory to process accesses in parallel.

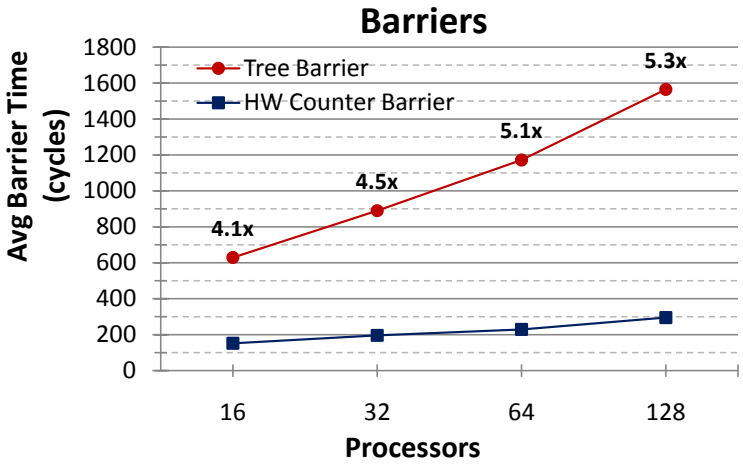


Figure 4.8: Average per processor latency across of the number of barrier episodes versus the number of processors, for simulated tree barriers using coherently cached variables and counter based barriers with automated notification signals.

Figure 4.8 shows the performance of the two barrier implementations. The counter-based barrier is from $4.1\times$ faster for 16 cores to $5.3\times$ faster for 128 cores. The former ($4.1\times$) is excessive compared to the expected $2\times$ and is probably due to the serialized injection of responses for the barrier end-signal, and due to the limited ability of the directory to process accesses in parallel. The latter ($5.3\times$), reflects the additive effect of the serialized injection for the barrier end-signal.

For both MCS barriers and locks, one can expect that aggressive non-blocking coherence protocols [4, 106] 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.

4.2.4 Summary

Counter-based barriers and multiple-reader queue based locks provide $3\times$ to $5\times$ better performance than state-of-the-art, highly-tuned, software-only barrier and lock implementations. The hardware cost of these synchronization primitives, presented in the previous chapter (subsection 3.2.6), may be justifiable, especially when

other uses of these primitives are taken into account, as those of chapter 2 (subsection 2.4.3). The next section evaluates the use of multiple-reader queues for task scheduling.

The evaluation of this section, although very promising, is only partial. The performance gain, by use of the proposed hardware assisted locks and barriers for synchronization, needs to be studied in larger benchmarks and applications, to assess how effective they can be.

4.3 Evaluation of Task Scheduling Support in the SARC Network Interface

This section evaluates the use of multiple-reader queues for scheduling parts of a larger computation (task scheduling). This is done by augmenting the runtime system for the Cilk [107] multithreading extensions to the C language, so that it utilizes multiple-reader queues instead of locks in the implementation of task scheduling.

A task (or job) is a part of a larger computation, that is intended to run in parallel with other tasks in a multiprocessor system. A task can be an instance of a function, a loop iteration (or a number of iterations), or simply a block of code. Tasks should be selected so that there are other independent tasks that can be executed in parallel. Alternatively, tasks can be selected so that they only have a few dependences with each other and can provide partial execution overlap. Task scheduling refers to tracking the generation of new tasks and selecting processors to run them. The latter part, selecting a processor to run a new task, is also called job-dispatching.

The following subsections explain how Cilk works and how it was augmented with multiple-reader queues, and present results from running Cilk applications. Some scalability issues for many-core processors are discussed then, and conclusions are summarized.

4.3.1 Cilk Background

Cilk implements multithreading extensions for the C language, which consists of three keywords to indicate parallelism and synchronization. A Cilk program run on one processor has the same semantics as the C program that results when the Cilk keywords are deleted. This C program is called the *serial elision* or the *C elision* of the Cilk program.

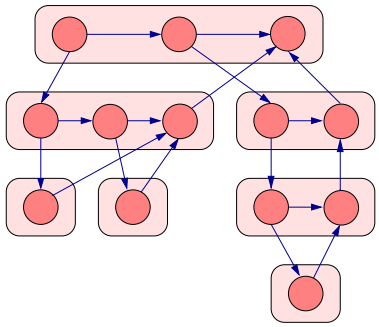


Figure 4.9: The Cilk model of multithreaded computation. Each procedure, shown as a rounded rectangle, is broken into sequences of threads, shown as circles. A downward edge indicates the spawning of a subprocedure. A horizontal edge indicates the continuation to a successor thread. An upward edge indicates the returning of a value to a parent procedure. All three types of edges are dependencies which constrain the order in which threads may be scheduled. (This figure is reconstructed from the Cilk language manual.)

The three basic Cilk keywords are **cilk**, **spawn**, and **sync**. The keyword **cilk** identifies a *Cilk procedure*, which is the parallel version of a C function –i.e. a function that can be run as a task on another processors. A Cilk procedure may spawn subprocedures in parallel and synchronize upon their completion. A Cilk procedure definition is identical to that of a C function, beginning with the keyword **cilk**.

Most of the work in a Cilk procedure is executed serially, just like C, but parallelism can be created when the invocation of a Cilk procedure is immediately preceded by the keyword **spawn**. A spawn is the parallel analog of a C function call, and like a C function call, when a Cilk procedure is spawned, execution proceeds to the child. Unlike a C function call, however, where the parent is not resumed until after its child returns, in the case of a Cilk spawn, the parent can continue to execute in parallel with the child. Indeed, the parent can continue to spawn off children, producing a high degree of parallelism.

A Cilk procedure cannot safely use the return values of the children it has spawned until it executes a **sync** statement. If all of its previously spawned children have not completed when it executes a **sync**, the procedure suspends and does not resume until all of those children have completed. The **sync** statement is a local "barrier", not a global one: **sync** waits only for the previously spawned children of the procedure to complete, and not for all procedures currently executing. As an aid to programmers, Cilk inserts an implicit **sync** before every return, if it is not present already. As a consequence, a procedure never terminates while it has outstanding children.

It is sometimes helpful to visualize a Cilk program execution as a directed acyclic graph, or dag (see figure 4.9). A Cilk program execution consists of a collection of procedures[5], each of which is broken into a sequence of nonblocking threads. In Cilk terminology, a *thread* is a maximal sequence of instructions that ends with a spawn, sync, or return (either explicit or implicit) statement. The first thread that executes when a procedure is called is the procedure's initial thread, and the subsequent threads are successor threads. At runtime, the binary "spawn" relation causes procedure instances to be structured as a rooted tree, and the dependencies among their threads form a dag embedded in this spawn tree. A correct execution of a Cilk program must obey all the dependencies in the dag, since a thread cannot be executed until all the threads on which it depends have completed.

Cilk is an algorithmic multithreaded language, meaning that the runtime system's scheduler guarantees provably efficient and predictable performance. Denote the execution time of a Cilk program with one processor T_1, with P processors T_P, and with an infinite number of processors T_∞. T_1 corresponds to the *work* of the computation, and is the time to execute all threads in the dag. T_∞ is called the *span* of the computation, and corresponds to the time needed to execute threads along the longest path of dependency in the dag. Similarly, denote S_1 the space needed for the execution of a Cilk program on one processor, and S_P the space needed for execution on P processors.

Two limits must hold for the execution time of a program on P processors: (i) $T_P \geq T_1/P$, and (ii) $T_P \geq T_\infty$. It is proven [108, 109] that a Cilk computation on P processors will require time $T_P \leq T_1/P + O(T_\infty)$, and space $S_P \leq S_1 P$. Empirically, the constand factor hidden by the big O is often 1 or 2 [1], and the expression

$$T_P \approx T_1/P + T_\infty$$

is a good approximation of execution time with P processors (for Cilk programs that do not use locks). Further, the notion of **parallelism** is defined as $\overline{P} = T_1/T_\infty$, and corresponds to the average amount of work for every step along the span. Whenever parallelism is much more than the number of processors used to execute a computation (i.e. $\overline{P} \gg P$), the relation $T_1/P \gg T_\infty$ will also hold, resulting in $T_P \approx T_1/P$.

Note, however, that in a many-core processor the number of cores may be more than the parallelism in the computation, and as a result approaching linear speedup may be infeasible. In this case, the span, which corresponds to the computation's critical path, dominates execution time, and two approaches are possible to achieve better speedup for well written parallel programs: (i) increase the exploitable parallelism of the given computation (with the same data-set) executing in

[5]Technically, procedure instances.

parallel smaller amounts of work, and (ii) increase parallelism using a larger data-set.

4.3.2 Cilk Scheduling and Augmentation with Multiple-Reader Queues

During the execution of a Cilk program, when a processor runs out of work, it "asks" another processor chosen at random for work to do. Locally, a processor executes procedures in ordinary serial order (just like C), exploring the spawn tree in a depth-first manner. When a child procedure is spawned, the processor saves local variables of the parent on the bottom of a stack and commences work on the child (the convention used is that the stack grows downward, and that items are pushed and popped from the bottom of the stack.) When the child returns, the bottom of the stack is popped (just like C) and the parent resumes. When another processor needs work, however, it steals from the top of the stack, that is, from the end opposite to the one normally used by the victim processor.

Two levels of scheduling are implemented in Cilk: nanoscheduling, which defines how a Cilk program is scheduled on one processor, and microscheduling, which schedules procedures across a fixed set of processors. The nanoscheduler guarantees that on one processor, when no microscheduling is needed, the Cilk code executes in the same order as the C code. This schedule is very fast and easy to implement. The **cilk2c** source-to-source compiler translates each Cilk procedure into a C procedure with the same arguments and return value. Each spawn is translated into its equivalent C function call. Each sync is translated into a no-operation, because, with nanoscheduler-managed serial execution, all children would have already completed by the time the sync point is reached.

In order to enable the use of the microscheduler, some code must be added to the nanoscheduled version of the code to keep track of the run-time state of a Cilk procedure instance. The nanoscheduler uses a double-ended queue of "frames" for this purpose. A Cilk *frame* is a data structure which can hold the run-time state of a procedure, and is analogous to a C activation frame. A doubly-ended queue, or *deque* in Cilk terminology, can be thought of as a stack from the nanoscheduler's perspective. There is one-to-one correspondence between Cilk frames on the deque and the activation frames on the C stack, which can be used in the future to merge the frame deque and the native C stack.

To keep track of the execution state at run-time, the nanoscheduler inserts three MACROs in every Cilk procedure. The first allocates a Cilk frame at the beginning

of the procedure. At the point a subprocedure's invocation return, a second macro checks the parent procedure's frame to determine if it has been stolen. The third macro just frees the Cilk frame of the procedure instance before returning to the caller. Some additional code is also introduced before subprocedure invocations, to store in the frame an index to a table of "return" labels for the procedure and any live variables at that point.

The microscheduler is implemented as a randomized work-stealing scheduler. Specifically, when a processor runs out of work, it becomes a *thief* and steals work from a *victim* processor chosen uniformly at random. When it finds a victim with some frames in its deque, it takes the topmost frame (the least recently pushed frame) on the victim's deque and places it in its own deque. It then gives the corresponding procedure to the nanoscheduler to execute.

The procedure given to the nanoscheduler is a different version of the stolen procedure, however. This version is called the "slow" version of the procedure, because it has a higher overhead than the normal nanoscheduled routine. Because the slow version is invoked from a generic scheduling routine, its interface is standard. Instead of receiving its arguments as C arguments, it receives a pointer to its frame as its only argument, from which it extracts its arguments and local state. It also produces a return value using a macro instead of returning a result normally. Finally, because this procedure may have outstanding children on another processor, it may suspend at a synchronization point, based on the return value of another macro.

Since slow versions of procedures are created only during a steal, their frame is always topmost in a processor's deque. All other procedures in the frame deque are the standard, fast, nanoscheduled versions. Therefore, as long as the number of steals is small, most of the computation is performed using the nanoscheduled routines.

In Cilk, operating system threads called *workers* are created before user program execution starts, based on a processor number command line parameter passed to the Cilk runtime system. Another parameter can specify if threads are bound or affine to processors, provided the OS supports it. In the simulation results of the next subsections, processor binding of Solaris lightweight processes is employed.

In the most recent Cilk version v5.4.6 a worker's deque does not store the Cilk frames one-by-one, but a single pointer to a *Closure* structure that includes the frame stack and additional information about the state of the worker. Thieves trying to steal from a worker's deque need to lock it first, and then also lock the closure found at the top. Thieves cannot steal from a closure containing a single frame, because the local worker is running on this frame. To reduce the overhead of the nanoscheduler,

an optimization is exploited, based on the observation that the local worker does not need to lock his deque or the closure, unless he tries to move the bottom pointer to the last stack frame and a thief simultaneously tries the same with the top pointer. An additional exception pointer, that indicates a thief's intention to steal *before* the actual steal, is used to signal this situation [107]. As a result a worker's deque has, at all times the whole frame stack at its top.

In order to optimize the locking process, a multiple-reader queue (mr-Q) replaces the deque, which under normal circumstances only buffers a single closure pointer [6]. This setting has the result of identifying the deque lock-token with the locked value (the closure pointer) of the original code. Because of this, two thieves trying to steal from each other would deadlock. To avoid this situation and prevent deadlock, thieves do not use the multiple-reader queue as a lock, but only as a queue providing atomicity of enqueue and dequeue operations (i.e. atomic update of head and tail pointers). This is done using unbuffered read messages to steal from the mr-Q (see subsection 2.4.2), which are NACK'ed if the mr-Q is empty. The local worker, though, uses the normal blocking read messages to access the multiple-reader queue when removing a closure, and as a result takes precedence over thieves in removing a closure from the mr-Q.

The augmentation of Cilk runtime to use multiple-reader queues, changes a property of the original implementation. Namely, thieves do not "queue-up" in front of a victim's dequeue when simultaneously targeting the same victim. This property was used in the proof of the execution time bound for the work-stealing schedule [108]. Nevertheless, subsequent research generalizes the execution time bound for non-blocking deques [110] that do not have this property. In addition, in the multiple-reader queue augmented scheduler, because thieves use non-blocking read messages and because only a single closure is placed in the mr-Q, the buffering required for each multiple-reader queue is restricted to one cache lines plus the queue descriptor ESL.

4.3.3 Cilk Results

The results of this section use the simulation infrastructure of subsection 4.2.2. Per processor intervals are measured for true work, overhead, steal time, and returning time of the scheduler loop. True work corresponds to the time of nanoscheduler's

[6] In the case of aborting computation using the additional **abort** keyword, it is possible that the worker places other, aborting closures at the bottom of his deque. The augmented version preserves the deque in software for this purpose. Aborting closures can not be stolen by thieves, as in the original code.

execution minus overhead time. Overhead is the time accounted to Cilk instrumentation of a benchmark's code with MACROs[7], described in subsection 4.3.2. Steal time is the time a worker's scheduler is searching for a victim, until a successful steal is performed. Whenever a steal occures, a (sub)procedure's execution continues in two processors. Because the microsheduler is involved and two closures are created for the (sub)procedure, when the stolen computation finishes it updates a list of descendants and any return value of a spawned child in the original closure. The time for these tasks is measured as returning time. The remaining time for scheduler loop execution is labeled other in the charts of this subsection. In addition several events are counted on a per processor bases.

Figure 4.10 shows the results of running the Cilk implementations of FFT, Cilksort, Cholesky, and LU benchmarks (provided with the Cilk distribution), using the original Cilk runtime (SW de-Q) and the multiple-reader queue augmented version (HW mr-Q). Cilk's recursive implementations of the FFT kernel, with a medium data set size of 1M complex numbers, provides scaling up to 64 processors. The mr-Q augmented runtime is about 20%-24% faster than the original for less than 128 processors, and 8% faster for 128 processors. Steal time is reduced in the mr-Q version by about 26%, 16%, 20% and 8% for 16, 32, 64 and 128 processors respectively. In addition, true work is reduced by 20% for 128 processors and 23-24% for other configurations. This means that the multiple-reader queue based scheduler exhibits better locality properties for FFT. Cilk overhead and returning time increase with more processors, and with 128 processors account in total for 18% and 26% of the execution time in the SW de-Q and HW mr-Q versions respectively. With 128 processors both versions slow down compared to the 64 processor execution.

Cilksort implements in Cilk's recursive style a variant of mergesort based on an algorithm that first appears in [111]. The executions shown in figure 4.10 sort an array of $3 \cdot 10^6$ integers and scale up to 64 processors. Although steal time is reduced with the HW mr-Q version in all configurations, performance of the two versions is always within 1% of each other. Cholesky executions, which use an 1000×1000 sparse matrix with 10000 non-zero elements, also do not scale beyond 64 processors with either version. The HW mr-Q version is 3-8% faster across processor numbersand steal time is reduced by 19-24%. The total of overhead and returning times on 16 and 32 processors remains below 2.5% of execution time for both scheduler versions, but for larger configurations it reaches 5-6% for the SW de-Q version and 10-14% for the HW mr-Q.

The LU kernel, with a medium data set of an 1024×1024 matrix, also scales

[7] The code saving values in a Cilk frame before a spawn was not accounted as overhead, and thus appears in the true work time.

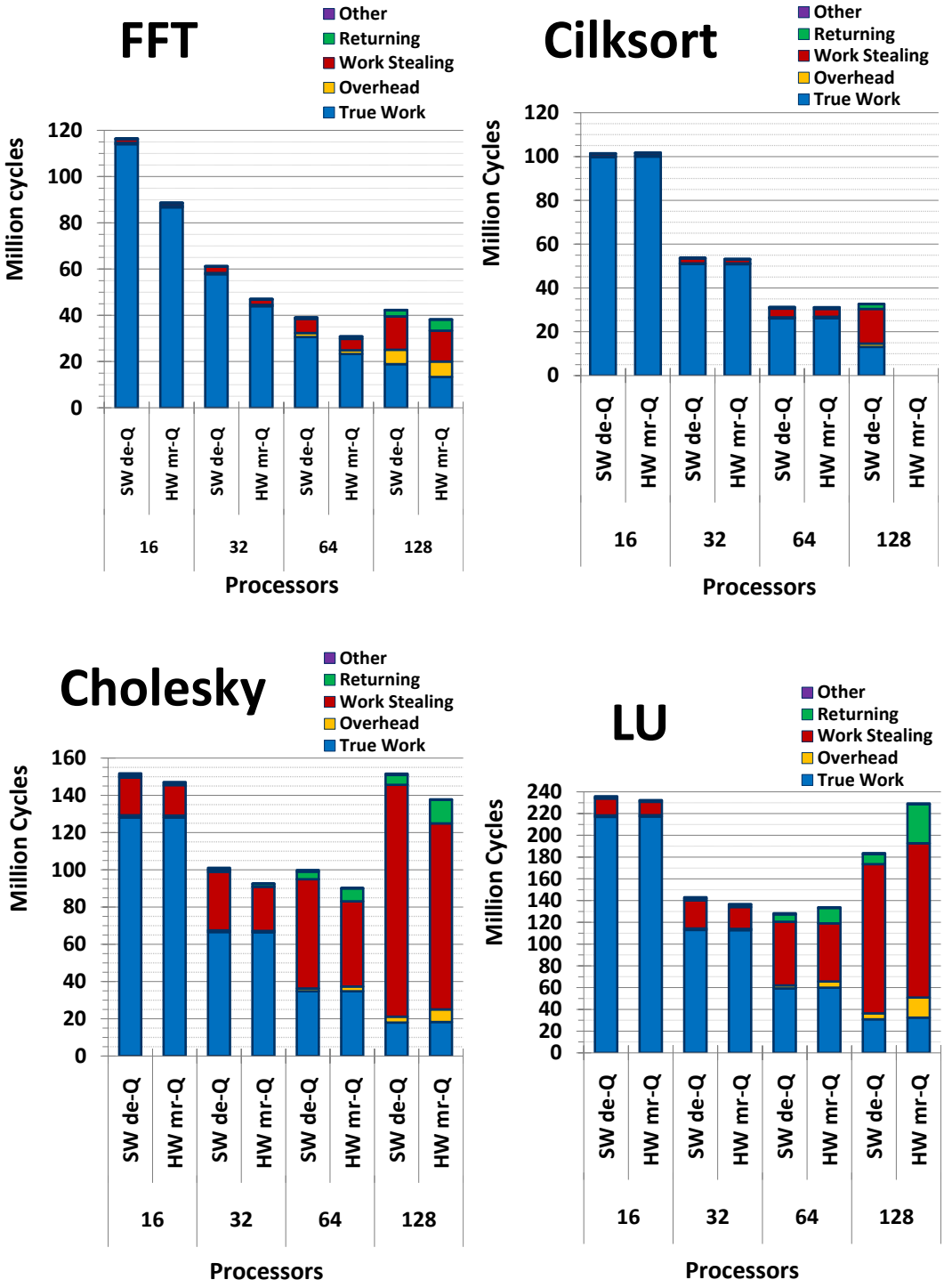


Figure 4.10: FFT, Cilksort, Cholesky, and LU benchmark execution with multiple-reader queue augmentation of the Cilk scheduler (HW mr-Q) and without (SW de-Q).

up to 64 processors with both scheduler versions. The HW mr-Q version is faster than the SW de-Q version, up to 32 processors. More specifically, the HW mr-Q version is about 1% and 4% faster, and steal time is reduced by 21% and 22% on 16 and 32 processors respectively. The HW mr-Q scheduler is about 7% and 31% slower than the SW de-Q scheduler, for 64 and 128 processors respectively. Steal time is only reduced by 9% on 64 processors with the HW mr-Q version, and it is increased by 3% with 128 processors. The total of overhead and returning times that is less than 2.5% of the execution time for both scheduler versions with 16 and 32 processors, increases to 14% and 22% for the HW mr-Q version with 64 and 128 processors respectively, where in the SW de-Q version it does not exceed 7-8%. Both versions provide some performance scaling with up to 64 processors and slow down with 128 processors.

The behavior of the HW mr-Q augmented scheduler for LU with 64 and 128 processors, as well as for FFT and cholesky (and cilksort) with 128 processors on both scheduler versions, is similar: overhead and returning times are increased, and steal time dominates the total execution time. This is because the large number of processors breaks down the total work in more parts, i.e. the total number of successful steals increases relative to that in smaller configurations, which results in more overhead and more returning time. Nevertheless, at some point the minimum work granularity is met (that imposed by recursion-end conditions), and altough some processors *still have work* to process, most of the others spends a long time in searching for work to steal but not finding any. These explanations are verified by work and steal interval event counts obtained with instrumentation. In the next subsection, the minimum work granularity is reduced for two of the benchmarks, in an effort to alleviate this load imbalance.

4.3.4 The Problem of Minimum Task Granularity

LU and cholesky are modified to reduce their minimum work granularity. For LU, the 1024×1024 array was broken in at most 64 blocks of $16x16$ elements (which explains why there is not enough work for 128 processors). The Cilk implementation is modified to use 256 blocks of 4×4 elements, to allow enough work for 128 processors. For cholesky, instead of ending the recursion for blocks of the original matrix smaller than 16×16, subprocedures are spawned down to 4×4 blocks. Cholesky and LU where chosen because the available parallelism plateaus at about 32 processors.

The results are shown in figure 4.11. Execution time increases for cholesky by about 90 million cycles, and for LU by about 135 million cycles for 16 processors.

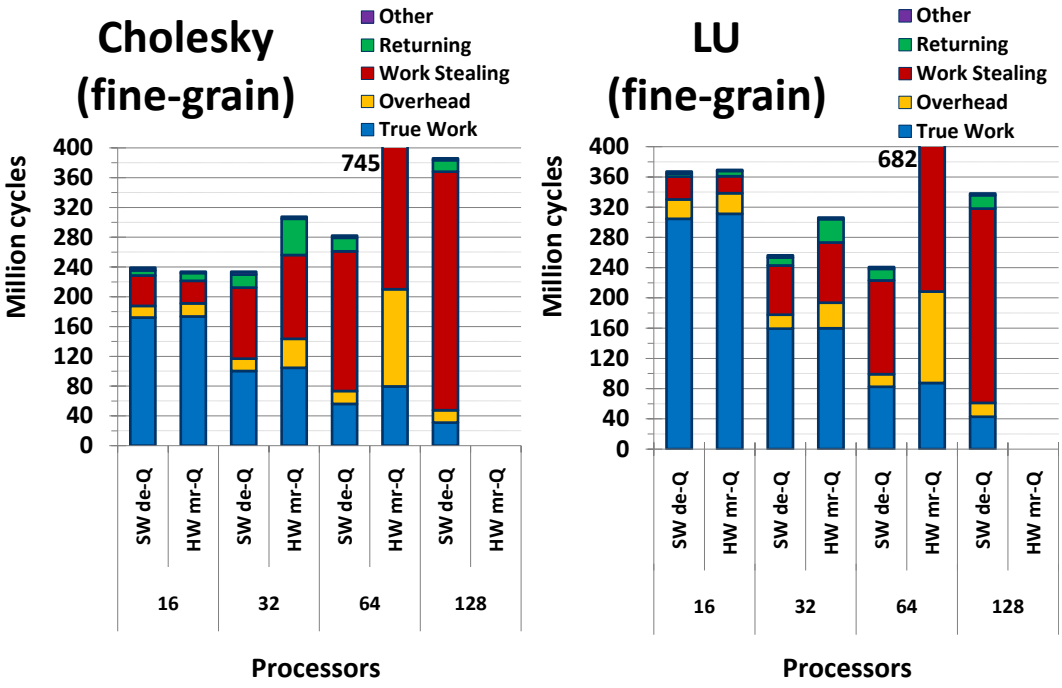


Figure 4.11: Cholesky, and LU benchmark execution, allowing fine granularity work partitioning.

Note that for LU performance of the original Cilk scheduler *appears* to provide scaling. In addition, steal, overhead, and returning times are multiplied in all configurations and for both versions of the scheduler. The performance of cholesky does not scale with more than 16 processors, and in both benchmarks and almost all configurations the original Cilk scheduler performs better than the multiple-reader queue augmented one (in fact, the HW mr-Q version simulations for 128 processors did not finish after more than a month). These results indicate that Cilk's overhead in work stealing makes it inappropriate for fine-grain processing.

Figure 4.11 shows a considerable increase in true work compared to the executions of figure 4.10. This increase indicates locality is an issue, for these CMP sizes and randomized work stealing of fine-grain tasks. Our event counts indicate that the resulting average true work per interval is more than $2 \cdot 10^4$ cycles in length, although the minimum is not available from these runs. The reason for the increase in true work is increased communication among thieves and victims with a large number of processors. Executing part of a computation in parallel requires communication and synchronization with both the producer of the computation's input and the consumer of the computation's output. If these times are relatively large, serial

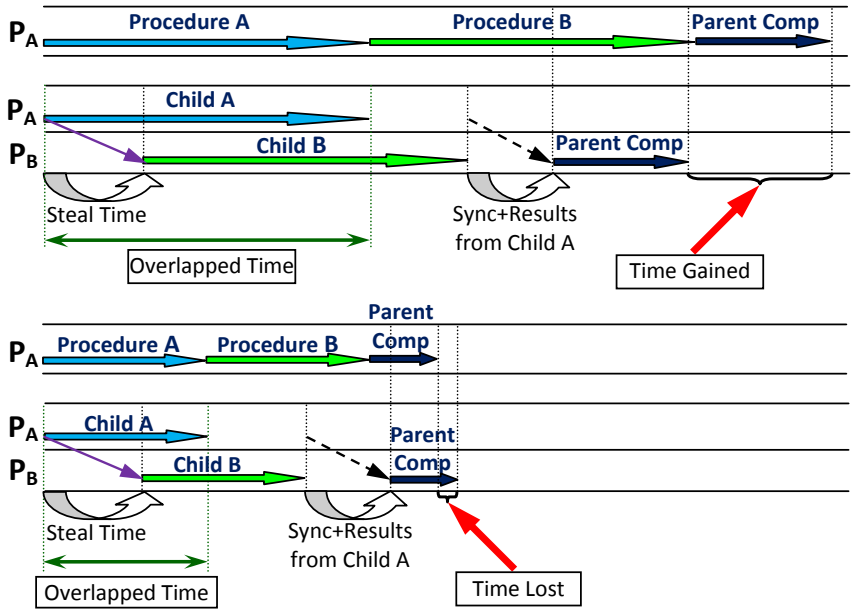


Figure 4.12: Illustration of the minimum granularity problem for a usual Cilk computation.

execution may be faster. In the context of Cilk, the producer is the victim of a steal attempt, the thief acts as a computation offload processing engine, and the consumer is either the victim or the thief –the one finishing last its computation.

Consider a typical Cilk procedure spawning two subprocedures and followed by some additional computation after a sync. In the upper part of figure 4.12, two alternative executions of such a Cilk procedure are illustrated. In the first, processor P_A executes serially the whole computation. In the second, P_A executes locally subprocedure A (child A) while P_B steals the parent procedure frame and starts execution of subprocedure B (child B). When P_B returns to the parent frame, queries Cilk runtime to find out if child A has finished. In the scenario shown child A has finished, so P_B copies its computation result in the parent frame and continues with the execution of the parent computation. The time gained by parallel execution of the two subprocedures is pointed by the red arrow on the right. The lower part of the figure shows the same execution, but this time assuming procedures A and B, as well as the parent computation take half the time, while the times for P_B to steal the parent frame, query Cilk runtime and copy the results of child A are the same. In this case, parallel execution of the two subprocedures takes more time than serial execution on P_A, as indicated by the red arrow on the right.

In both cases, the time overlap of processors P_A and P_B starts when P_B initiates its steal attempt, and finishes when P_A is done with the execution of his part of the

computation (i.e. child A). If this overlapped time is more than the time required by P_B to complete the steal plus the time to synchronize with P_A through Cilk runtime and get the result of child A, then there is a positive time gain from parallel execution. Otherwise, the time required in excess of the overlapped time for these communication and synchronization actions with P_A, is time lost compared to serial execution on P_A.

4.3.5 Summary

Evaluation of the use of multiple-reader queues in augmentation of the Cilk runtime system simulating 16-128 processors, provides three results. First, for parallelization in this range of processors on a single-CMP environment, the minimum task granularity is very important, in spite of scalable behavior for up to 64 processors. In fact, with the benchmarks used, selecting an inappropriately small minimum task granularity results in larger execution time of the "best-performing" 64 processor execution, than that of only 16 processors with a better minimum task size selection. This is exaggerated with hardware support that slices work to pieces faster, and results in bad scaling behavior as more processors are used.

Second, regardless of the choice of minimum task granularity, the benchmarks used (FFT, LU, Cholesky and Cilksort) do not scale over 64 processors. With appropriate selection of minimum task granularity and 16-64 processors, hardware support for scheduling provides 20.9%-23.9% better performance of regular codes than the original Cilk without hardware support. For irregular parallelism, though, the performance gain is reduced to 0.2%-9.5% and the version exploiting hardware support may scale to fewer processors than the software-only version (though this does not necessarily correspond to top performance reduction). Third, in 64 processor executions, the average true work interval (i.e. task size without overheads) is always more than 30 thousand clock cycles in simulations of this section, and overheads range across benchmarks to about 20%-60% of average task size. This indicates that Cilk cannot be effective for fine-grain parallelization in this range of processors, even if minimum task granularity was chosen correctly and localized work stealing was implemented in the runtime.

Exploiting hardware support for fine-grain job scheduling to a large number of processors in the context of Cilk is not trivial. The problem of selecting the minimum task granularity, that Cilk leaves to the application, may result in increased execution times and reverse performance scaling when more processors are used.

Collaborators: Michael Zampetakis contributed the results of subsection 4.1.2 from running several versions of STREAM, FFT, and Bitonic on the hardware prototype.

5

Conclusions

5.1 Conclusions

This dissertation demonstrates the integration of a network interface with a cache controller, that allows configurable use of processor-local memory, partly as cache and partly as scratchpad. Chapter 3 shows that NI integration can be done efficiently, requiring less than 20% logic increase of a simple cache controller and one or two more state bits per cache line. The novel technique of event responses is introduced, which enables this efficient cache-integration of NI mechanisms. However, hardware cost assessment indicates, that support for synchronization primitives may not be cost-effective for general purpose deployment, at least on a per core basis.

Albeit only partially evaluated here, this thesis presents the design and implementation of a general RDMA-copy mechanism for a shared address space, and synchronization counter support for selective fences of arbitrary groups of explicit transfers or barriers. Similarly, it demonstrates the design and implementation of single-reader queues, aiming to enhance many-to-one interaction efficiency.

In addition, multiple-reader queues are introduced, as a novel synchronization primitive that enables “blind” rendez-vous of requests with replies, and their matching in the memory system, without processor involvement. Chapter 4 shows that the explicit transfers and the synchronization primitives, provided by the cache-integrated NI, offer flexibility via latency- and bandwidth-efficient communication mechanisms, or allowing mixed use of implicit and explicit communication, and also benefit latency-critical tasks like synchronization and scheduling.

Looking back for lessons learned from the course of the research for this dissertation, there are two specific things to mention, related to the evaluation of scheduling support in the SARC NI. First, the Cilk-based study of section 4.3, indicates that flat randomized work stealing is probably not a good idea with several tens of processors. Some form of hierarchical extention of this scheduling scheme is required, in favor of locality.

Second, although this is not new, the results of subsection 4.3.4 underline the fact that scalability in itself is neither a suitable nor an appropriate metric for parallel execution optimizations. Changes in the problem size or the granularity of parallelism exploited, may uncover that a scalable behavior is very far from optimal. Thus, some method is required to determine the optimal behavior, especially when, using single processor execution as a reference point, is not convenient or possible.

Above other lessons, though, I must underline my view on setting *early* a concrete motivation for research, deduced from the course of this study. Novel ideas,

that may seem promising, but only in an abstract sense or view, must be preliminarily evaluated within a limited time frame, and possibly before completing their design or implementation, so that they are placed in a framework of solving a perceived actual problem. This is most important –and more likely to occur– for basic research that may give rise to radical changes. Moreover, it means, that methods to achieve such preliminary evaluation, should not be under *interdependent* research. Reversely, it may be even better to have the ground of solving a specific actual problem as the starting point for innovation. In the field of computer architecture, confining research motivation to improvement upon a tangible problem, is established and defended by the international culture developed around quantitative research assessment.

5.2 Future Perspective

Research efforts during this study and in the CARV laboratory of FORTH-ICS, identified at least three important issues, that in the author's perspective deserve further research. The first pertains generally to run-time systems, and the other two are mostly oriented to the exploitation of explicit communication in CMPs.

The first issue concerns the need for locality-aware scheduling of computations in large-scale CMPs, and the associated need for management, possibly at run-time, of the minimum profitable task size. It is becoming increasingly apparent, that, with tens of processors on a single chip, locality exploitation will play a critical role. In addition, the discussion of the minimum task granularity problem in subsection 4.3.4, shows the importance of determining, for a given computation, the amount of parallelism and the number of cores that can be exploited profitably, on the given hardware.

The second issue is related to the need for methodologies and mechanisms, associated with software-managed on-chip memory, that enable the manipulation of working sets that do not fit in on-chip storage, and automate spilling and re-fetching of scratchpad data. In a broader context, novel hardware support for flexible and software-configurable locality management, can be of interest to both cache- and scratchpad-based parallel processing.

The third issue refers to the broad exploitation of producer-initiated explicit transfers, that could maximize the benefits of explicit communication. There are at least two potentially problematic aspects in the general use of such transfers. Namely, the need to know in advance where the consumer of produced data is or

will be scheduled, and the management of consumer local memory space, discussed in subsection 2.3.6.

Although an almost ten year time frame has gone by, exploitation of multiple cores in a general computation is an open problem, that seems to affect the course of the general purpose processor industry. In the beginning of this study, software targeting such computations was very scarce. The lack of software benchmark sets for the evaluation of new architectures, has been an obstacle to the development of appropriate hardware support, and will also be in the near future.

At this time, many research efforts propose extended [17, 18, 19], or novel [112, 93] programming model semantics, but either evaluate their success for less than 16 cores, or specifically target processing of regular streams of data and data-parallel applications. In any case, there seems to be a consensus toward the breakdown of a computation in lightweight tasks or kernels, manipulating their parallel processing under the orchestration of a compiler-generated run-time system, and with the aid of higher-level or library constructs.

Because mapping a general computation on specific hardware resources and dynamic scheduling are too complex for the end-programmer, run-time systems are essential and should provide a central software platform for evaluation of multicore hardware support. For example, run-time systems and libraries can exploit hardware support for synchronization and explicit communication, like that advocated here, as in the case of the Cilk-based evaluation of section 4.3.

However, it is the author's opinion, that to enable new parallel software, the programmer should not be confined to the serial semantics of uniprocessor programming languages. New, higher level, programming model semantics are required to program groups of processors, that will urge the programmer to expose the maximum possible parallelism in an application. Exposing language defined tasks and data structures, as well as providing syntactic sugar for their recursive and repetitive declaration, can be examples of such higher level semantics in the programming model.

Parallel processing was abandoned in the past because of the difficulty of writing parallel programs. Contemporary efforts for general exploitation of multicore processors are also largely dependent on the progress of parallel software technology. It is doubtful if 65nm and 45nm integration processes are shelling as many processor chips as previous ones. This time it seems that computer engineering, and more generally computer science, will either get over the child-phase of uniprocessing in its history, or it will shrink in usefulness and importance. This may mean that interesting times are still ahead, in the future of computers.

Bibliography

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995.

[2] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[3] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: the end of the road for conventional microarchitectures," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 248–259, 2000.

[4] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: a scalable architecture based on single-chip multiprocessing," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2000, pp. 282–293.

[5] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson, "A 48-core ia-32 message-passing processor with dvfs in 45nm cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, Feb. 2010, pp. 108 –109.

[6] Intel, "Intel Unveils 32-core Server Chip at International Supercomputing Conference," http://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm, May 31 2010.

[7] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, no. 5, pp. 15 –31, 2007.

[8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[9] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," *SIGPLAN Not.*, vol. 37, no. 10, pp. 211–222, 2002.

[10] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A nuca substrate for flexible cmp cache sharing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, pp. 1028–1040, 2007.

[11] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive nuca: near-optimal block placement and replication in distributed caches," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2009, pp. 184–195.

[12] D. Pham et al., "The design and implementation of a first-generation CELL processor," in *Proc. IEEE Int. Solid-State Circuits Conference (ISSCC)*, Feb. 2005.

[13] 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 & Development.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[14] A. M. Aji, W.-c. Feng, F. Blagojevic, and D. S. Nikolopoulos, "Cell-swat: modeling and scheduling wavefront computations on the cell broadband engine," in *CF '08: Proceedings of the 5th conference on Computing frontiers*. New York, NY, USA: ACM, 2008, pp. 13–22.

[15] B. Gedik, R. R. Bordawekar, and P. S. Yu, "Cellsort: high performance sorting on the cell processor," in *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 1286–1297.

[16] S. Schneider, J.-S. Yeom, and D. S. Nikolopoulos, "Programming multiprocessors with explicitly managed memory hierarchies," *Computer*, vol. 42, no. 12, pp. 28–34, 2009.

[17] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 83.

[18] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta, "Extending the OpenMP tasking model to allow dependent tasks," in *IWOMP'08: Proceedings of the 4th international conference on OpenMP in a new era of parallelism*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 111–122.

[19] B. R. Gaster, "Streams: Emerging from a shared memory model." in *IWOMP*, ser. Lecture Notes in Computer Science, R. Eigenmann and B. R. de Supinski, Eds., vol. 5004. Springer, 2008, pp. 134–145. [Online]. Available: http://dblp.uni-trier.de/db/conf/iwomp/iwomp2008.html#Gaster08

[20] J. Duato, S. Yalamanchili, and N. Lionel, *Interconnection Networks: An Engineering Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[21] J. Duncanson, "Inverse multiplexing," *Communications Magazine, IEEE*, vol. 32, no. 4, pp. 34 –41, apr 1994.

[22] WP5 partners, under M. Katevenis coordination, "D5.6: Network Interface Evaluation, Prototyping, and Optimizations," Institute of Computer Science, FORTH, Heraklion, Greece, Tech. Rep. Confidential SARC Deliverable D5.6, May 2010.

[23] D. S. Henry and C. F. Joerg, "A tightly-coupled processor-network interface," *SIGPLAN Not.*, vol. 27, no. 9, pp. 111–122, 1992.

[24] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li, "Protected, user-level dma for the shrimp network interface," in *HPCA '96: Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 1996, p. 154.

[25] E. Markatos and M. Katevenis, "User-level DMA without operating system kernel modification," in *HPCA'97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture, San Antonio, TX USA*, Feb. 1997, pp. 322–331.

[26] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture," in *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*. New York, NY, USA: ACM, 1998, pp. 398–407.

[27] V. Papaefstathiou, G. Kalokairinos, A. Ioannou, M. Papamichael, G. Mihelogiannakis, S. Kavadias, E. Vlachos, D. Pnevmatikatos, and M. Katevenis, "An fpga-based prototyping platform for research in high-speed interprocessor communication," 2nd Industrial Workshop of the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC), October 17 2006.

[28] V. Papaefstathiou, D. Pnevmatikatos, M. Marazakis, G. Kalokairinos, A. Ioannou, M. Papamichael, S. Kavadias, G. Mihelogiannakis, and M. Katevenis, "Prototyping efficient interprocessor communication mechanisms," in *Proc. IEEE International Symposium on Systems, Architectures, Modeling and Simulation (SAMOS2007)*, July 16-19 2007.

[29] M. Papamichael, "Network Interface Architecture and Prototyping for Chip and Cluster Multiprocessors," Master's thesis, University of Crete, Heraklion, Greece, 2007, also available as ICS-FORTH Technical Report 392.

[30] H. Grahn and P. Stenstrom, "Efficient strategies for software-only directory protocols in shared-memory multiprocessors," in *Computer Architecture, 1995. Proceedings. 22nd Annual International Symposium on*, June 1995, pp. 38–47.

[31] D. Chaiken and A. Agarwal, "Software-extended coherent shared memory: performance and cost," *SIGARCH Comput. Archit. News*, vol. 22, no. 2, pp. 314–324, 1994.

[32] F. C. Denis, D. A. Khotimsky, and S. Krishnan, "Generalized inverse multiplexing of switched atm connections," in *Proceedings of the IEEE Conference on Global Communications (GlobeCom '98)*, 1998, pp. 3134–3140.

[33] D. A. Khotimsky, "A packet resequencing protocol for fault-tolerant multipath transmission with non-uniform traffic splitting," in *Proc. Global Telecommunication Conference (GLOBECOM)*, December 1999, pp. 1283–1289.

[34] E. Brewer, F. Chong, L. Liu, S. Sharma, and J. Kubiatowicz, "Remote queues: Exposing message queues for optimization and atomicity," in *Proc. 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA'95), Santa Barbara, CA USA*, Jun. 1995, pp. 42–53.

[35] S. Mukherjee, B. Falsafi, M. Hill, and D. Wood, "Coherent network interfaces for fine-grain communication," in *Proc. 23rd Int. Symposium on Computer Architecture (ISCA'96), Philadelphia, PA USA*, May 1996, pp. 247–258.

[36] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-net: a user-level network interface for parallel and distributed computing," in *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1995, pp. 40–53.

[37] S. L. Scott, "Synchronization and communication in the t3e multiprocessor," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 5, pp. 26–36, 1996.

[38] Y. Osborne, R. Osborne, Q. Zheng, Q. Zheng, J. H. Merl, J. H. Merl, R. Casley, R. Casley, D. Hahn, and D. Hahn, "Dart - a low overhead atm network interface chip," in *in Proc. Hot Interconnects*, 1996, pp. 587–593.

[39] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, "The virtual interface architecture," *IEEE Micro*, vol. 18, no. 2, pp. 66–76, 1998.

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

[41] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The stanford dash multiprocessor," *Computer*, vol. 25, no. 3, pp. 63–79, 1992.

[42] U. Ramachandran, G. Shah, A. Sivasubramaniam, A. Singla, and I. Yanasak, "Architectural mechanisms for explicit communication in shared memory multiprocessors," in *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM, 1995, p. 62.

[43] D. Koufaty and J. Torrellas, "Comparing data forwarding and prefetching for communication-induced misses in shared-memory mps," in *ICS '98: Proceedings*

of the 12th international conference on Supercomputing. New York, NY, USA: ACM, 1998, pp. 53–60.

[44] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," in *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, pp. 434–442.

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

[46] C. Villavieja, M. Katevenis, N. Navarro, D. Pnevmatikatos, A. Ramirez, S. Kavadias, V. Papaefstathiou, D. Nikolopoulos, "Hardware Support for Explicit Communication in Scalable CMP's," Computer Architecture Dept., Polythecnic University of Catalonia (UPC), Barcelona, July 2008 version, Tech. Rep. Technical Report UPC-DAC-RR-CAP-2009-1, 2008. [Online]. Available: http://gsi.ac.upc.edu/reports/2009/1/LMnDMAvsCnPP_2008jul.pdf

[47] S. Borkar, R. Cohn, G. Cox, S. Gleason, and T. Gross, "iWarp: an integrated solution of high-speed parallel computing," in *Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1988, pp. 330–339.

[48] W. J. Dally, J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, P. R. Nuth, R. E. Davison, and G. A. Fyler, "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms," *IEEE Micro*, vol. 12, no. 2, pp. 23–39, 1992.

[49] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee, "Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor," *SIGARCH Comput. Archit. News*, vol. 26, no. 3, pp. 306–317, 1998.

[50] U. Kapasi, W. J. Dally, S. Rixner, J. D. Owens, and B. Khailany, "The Imagine stream processor," in *Proceedings 2002 IEEE International Conference on Computer Design*, Sep. 2002, pp. 282–288.

[51] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2010, pp. 311–322.

[52] K. Mackenzie, J. Kubiatowicz, M. Frank, W. Lee, V. Lee, A. Agarwal, and M. Kaashoek, "Exploiting two-case delivery for fast protected messaging," in *HPCA*

'98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture. Washington, DC, USA: IEEE Computer Society, 1998, p. 231.

[53] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1992, pp. 256–266.

[54] T. Shimizu, T. Horie, and H. Ishihata, "Low-latency message communication support for the ap1000," in *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1992, pp. 288–297.

[55] "SARC: Scalable computer ARChitecture," http://www.sarc-ip.org/, 2005-2009, european IP Project.

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

[57] S. Kavadias, M. G. H. Katevenis, M. Zampetakis, and D. S. Nikolopoulos, "On-chip Communication and Synchronization with Cache-Integrated Network Interfaces," in *CF '10: Proceedings of the 7th ACM conference on Computing Frontiers*. New York, NY, USA: ACM, May 2010.

[58] D. Mosberger, "Memory consistency models," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 1, pp. 18–26, 1993, see also updated version as University of Arisona technical report TR 93/11.

[59] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, 1979.

[60] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta, "Integration of message passing and shared memory in the Stanford FLASH multiprocessor," *SIGOPS Oper. Syst. Rev.*, vol. 28, no. 5, pp. 38–50, 1994.

[61] Y. Chen, A. Bilas, S. N. Damianakis, C. Dubnicki, and K. Li, "Utlb: a mechanism for address translation on network interfaces," in *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 1998, pp. 193–204.

[62] I. Schoinas and M. Hill, "Address translation mechanisms in network interfaces," in *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 1998, p. 219.

[63] M. Katevenis, "Interprocessor communication seen as load-store instruction generalization," in *The Future of Computing, essays in memory of Stamatis Vassiliadis*, K. B. e.a., Ed., Delft, The Netherlands, Sep. 2007, pp. 55–68.

[64] P. Buonadonna and D. Culler, "Queue pair ip: a hybrid architecture for system area networks," 2002, pp. 247 –256.

[65] InfiniBand Trade Association, "Infiniband architecture specification release 1.2.1," http://www.infinibandta.org/specs/register/publicspec/, January 2008.

[66] R. Huggahalli, R. Iyer, and S. Tetrick, "Direct cache access for high bandwidth network i/o," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 50–59.

[67] J. Beecroft, D. Addison, D. Hewson, M. McLaren, D. Roweth, F. Petrini, and J. Nieplocha, "Qsnetii: Defining high-performance network design," *IEEE Micro*, vol. 25, no. 4, pp. 34–47, 2005.

[68] E. W. Dijkstra, *Cooperating Sequential Processes*. New York, NY, USA: Springer-Verlag New York, Inc., 2002, pp. 65–138.

[69] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[70] M. P. Herlihy, "Impossibility and universality results for wait-free synchronization," in *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1988, pp. 276–290.

[71] M. Byrd, G.T. Flynn, "Producer-consumer communication in distributed shared memory multiprocessors," *Proceedings of the IEEE*, vol. 87, no. 3, pp. 456–466, Mar. 1999.

[72] N. Aboulenein, J. R. Goodman, S. Gjessing, and P. J. Woest, "Hardware support for synchronization in the scalable coherent interface (sci)," in *Proceedings of the 8th International Symposium on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 1994, pp. 141–150.

[73] U. Ramachandran and J. Lee, "Cache-based synchronization in shared memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 32, no. 1, pp. 11–27, 1996.

[74] G. T. Byrd and B. Delagi, "Streamline: Cache-based message passing in scalable multiprocessors," in *ICPP (1)*, 1991, pp. 251–254.

[75] M. G. Katevenis, V. Papaefstathiou, S. Kavadias, D. Pnevmatikatos, F. Silla, and D. S. Nikolopoulos, "Explicit communication and synchronization in sarc," *IEEE Micro*, 2010, accepted for publication.

[76] J. Kubiatowicz and A. Agarwal, "Anatomy of a message in the Alewife multiprocessor," in *ICS '93: Proceedings of the 7th international conference on Supercomputing*. New York, NY, USA: ACM, 1993, pp. 195–206.

[77] S. C. Woo, J. P. Singh, and J. L. Hennessy, "The performance advantages of integrating block data transfer in cache-coherent multiprocessors," in *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 1994, pp. 219–229.

[78] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim, "Integrating message-passing and shared-memory: early experience," *SIGPLAN Not.*, vol. 28, no. 7, pp. 54–63, 1993.

[79] J. D. Kubiatowicz, "Integrated Shared-Memory and Message-Passing Communication in the Alewife Multiprocessor," Ph.D. dissertation, Massachusetts Institute of Technology, 1998, supervisor-Agarwal, Anant.

[80] P. Bannon, "Alpha 21364: A scalable single-chip smp," in *Eleventh Ann. Microprocessor Forum, MicroDesign Resources*, Sebastopol, California, 1998.

[81] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Cascaval, J. Castanos *et al.*, "An overview of the blue gene/l supercomputer," in *In Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, November 2002, p. 122.

[82] M. Reilly, L. C. Stewart, J. Leonard, and D. Gingold, "A new generation of cluster interconnect," http://www.sicortex.com/resource_center/white_papers/sicortex_technical_summary, December 2006/revised February 2009.

[83] S. Mukherjee and M. Hill, "A survey of user-level network interfaces for system area networks," Univ. of Wisconsin, Madison, USA, Computer Sci. Dept. Tech. Report 1340, 1997.

[84] R. A. Bhoedjang, T. Ruhl, and H. E. Bal, "User-level network interface protocols," *Computer*, vol. 31, no. 11, pp. 53–60, 1998.

[85] 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*. New York, NY, USA: ACM, 2000, pp. 214–224.

[86] IBM, *PowerPC 750GX/FX Cache Programming*, Dec 2004. [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/0DD2C54EDDF7EB9287256F3F00592C64

[87] Intel, *Intel XScale Microarchitecture Programmers Reference Manual*, Feb 2001. [Online]. Available: http://download.intel.com/design/intelxscale/27343601.pdf

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

[89] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Fine-grain access control for distributed shared memory," *SIGPLAN Not.*, vol. 29, no. 11, pp. 297–306, 1994.

[90] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and typhoon: user-level shared memory," *SIGARCH Comput. Archit. News*, vol. 22, no. 2, pp. 325–336, 1994.

[91] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood, "Application-specific protocols for user-level shared memory," in *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 380–389.

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

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

[94] H. Cook, K. AsanoviÄ‡, and D. A. Patterson, "Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-131, Sep 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-131.html

[95] 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*, vol. 28, no. 4, pp. 51–70, 2008.

[96] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing memory systems for chip multiprocessors," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 358–368, 2007.

[97] S. W. Keckler, A. Chang, W. S. Lee, S. Chatterjee, and W. J. Dally, "Concurrent event handling through multithreading," *IEEE Trans. Comput.*, vol. 48, no. 9, pp. 903–916, 1999.

[98] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *SIGARCH Comput. Archit. News*, vol. 18, no. 3a, pp. 364–373, 1990.

[99] G. Kalokerinos, V. Papaefstathiou, G. Nikiforos, S. Kavadias, M. Katevenis, D. Pnevmatikatos, and X. Yang, "Prototyping a Configurable Cache/Scratchpad Memory with Virtualized User-Level RDMA Capability," *Transactions on HiPEAC*, 2010, submitted for publication.

[100] H. Shan and J. P. Singh, "A Comparison of MPI, SHMEM and Cache-Coherent Shared Address Space Programming Models on a Tightly-Coupled Multiprocessors," *International Journal of Parallel Programming*, vol. 29, no. 3, pp. 283–318, 2001.

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

[102] Saman P. Amarasinghe and Michael I. Gordon and Michal Karczmarek and Jasper Lin and David Maze and Rodric M. Rabbah and William Thies, "Language and Compiler Design for Streaming Applications," *International Journal of Parallel Programming*, vol. 33, no. 2-3, pp. 261–278, 2005.

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

[104] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[105] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[106] K. Gharachorloo, M. Sharma, S. Steely, and S. Van Doren, "Architecture and design of alphaserver gs320," *SIGPLAN Not.*, vol. 35, no. 11, pp. 13–24, 2000.

[107] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1998, pp. 212–223.

[108] R. D. Blumofe and C. E. Leiserson, “Scheduling Multithreaded Computations by Work Stealing,” in *In Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, 1994, pp. 356–368.

[109] ——, “Scheduling Multithreaded Computations by Work Stealing,” *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[110] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, “Thread scheduling for multiprogrammed multiprocessors,” in *SPAA ’98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1998, pp. 119–129.

[111] S. G. Akl and N. Santoro, “Optimal parallel merging and sorting without memory conflicts,” *IEEE Trans. Comput.*, vol. 36, no. 11, pp. 1367–1369, 1987.

[112] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, “A stream compiler for communication-exposed architectures,” in *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2002, pp. 291–303.