INTERPROCESSOR COMMUNICATION SEEN AS LOAD-STORE INSTRUCTION GENERALIZATION

Manolis G.H. Katevenis[†]

Institute of Computer Science, Foundation for Research and Technology - Hellas (FORTH-ICS),
Vassilika Vouton, Heraklion, Crete, GR-70013 Greece - Member of HiPEAC
http://archvlsi.ics.forth.gr/∼kateveni/

ABSTRACT

This paper presents the current (2007) author's views and opinions on interprocessor communication (IPC) and how it should evolve in future multiprocessors, with an attempt to define an IPC architecture that is uniformly extensible from small-scale chip multiprocessors (CMP) to large-scale multi-chip parallel systems. We adopt the viewpoint that all IPC is based on data transfers through an (on-chip or multi-chip) interconnection network; all processors, acceleration engines, and independent memories interface to this network through *lightweight* and *tightly-coupled* network interfaces (NI). Just like traditional (RISC) processor-to-memory communication is via load and store instructions and atomic operation primitives, future IPC should be based on the corresponding data transfer primitives: remote read DMA, remote write DMA, and remote queues. We assume a global address space. By examining the traditional functions of address translation and network routing (and perhaps even caching), we observe that these are closely related, and thus will probably have to be provided through a unified mechanism, whose purpose should be to support data *migration* and *protection*; flow control and congestion management may have to be closely linked to this same mechanism. Two interesting possibilities are to architect NI's so that they detect some events and cause corresponding actions, and to implement cache coherence, when and where provided, on top of the above hardware primitives provided by the NI's.

1. INTERPROCESSOR COMMUNICATION IN PAST AND FUTURE MP'S

Digital processors increasingly work in cooperation, rather than in isolation. Chip multiprocessors (CMP) [1] [2] [3] contribute centrally to this trend. *Interprocessor Communication (IPC)* is the means by which processor cooperation is implemented. The purpose of this paper is to reflect upon future IPC architectures and to provide this author's opinion about them, even if parts of this opinion are based on intuition.

Past systems provided either *fast but non-scalable* IPC through coherent caches, or *scalable but long-latency* IPC through general-purpose interconnection networking. There was an underlying –sometimes in-admitted– assumption that large-scale *and* high-performance IPC was either not needed or not feasible or prohibitively expensive. This paper opposes that view, and tries to outline a *unifying* architecture for high performance IPC both at the small and at the large scale.

1.1. Communication to Computation Ratio

The ratio of (remote) communication to (local) computation varies widely from application to application, just like, e.g., some applications perform no floating-point calculations at all while other applications

[†]also with the Dept. of Computer Science, University of Crete, Greece

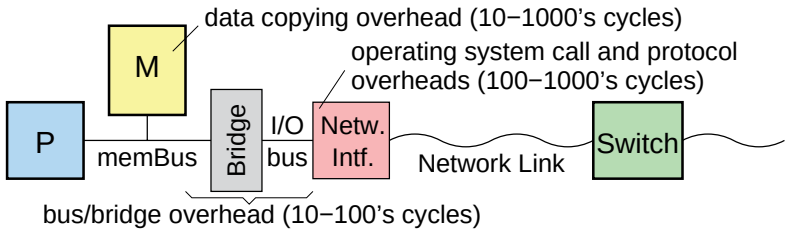

Fig. 1. Network Interfaces of the past and their overheads

critically depend on fast floating-point hardware for their performance. Interprocessor communication was expensive, relative to local computation, in the computer systems of the past half century, especially in large-scale multiprocessors (MP's larger than typical cache-coherence diameters). Thus developers architected their applications so as to avoid frequent or extensive communication, and so as to batch communicated data into coarse-grain units.

Some designers observe this property of typical current applications, and conclude that there is no need to support high ratios of communication-to-computation, or fine-grain communication patterns –or at least not so for communication by means other than coherent caches. Such an attitude, however, clearly creates a chicken-and-egg effect: applications will not use extensive, fine-grain communication as long as architectures do not support it, and new architectures will not support it because applications do not use it! In this author's opinion, new architectures should not omit or delay the exploitation of opportunities created by the new technology in the area of high-throughput, low-latency, and fine-grain communication, at all scales –both on-chip and system-wide. Such architectural support for advanced IPC will enable a whole new breed of parallel applications.

1.2. IPC Overheads in Cluster Computing using Network Interfaces of the Past

Large-scale multiprocessing is based, since a decade ago, on cluster computing. In clusters, interprocessor communication (IPC) has to occur through the I/O subsystem, which was designed for the peripheral devices of the past that were much slower than processors or modern interconnection networks. Figure 1 illustrates this placement of the network interface (NI) and its associated overheads; these overheads are listed below together with the methods to eliminate them, based on the research of the last 15 years. Advanced architectures have adopted some such solutions, and widespread adoption must occur in all forthcoming architectures.

- Each I/O operation required the intervention of the operating system in order to provide virtualization of the I/O devices, i.e. transparent sharing of these devices among multiple user processes. Traditional system call overhead is in the hundreds to thousands of clock cycles. To eliminate this, multiple methods for (protected) user-level access to the I/O devices have been demonstrated [4].

- I/O transfers occurred over the I/O bus, with latencies in the tens to hundreds of processor cycles –quite slower than the memory bus. To reduce this overhead, researchers have proposed to tightly couple the network interface to the cache hierarchy [5].

- I/O data had to be copied multiple times –typically at least once between network interface and memory, and once between kernel and user address space. To eliminate this overhead, zero-copy communication protocols have been developed [6].

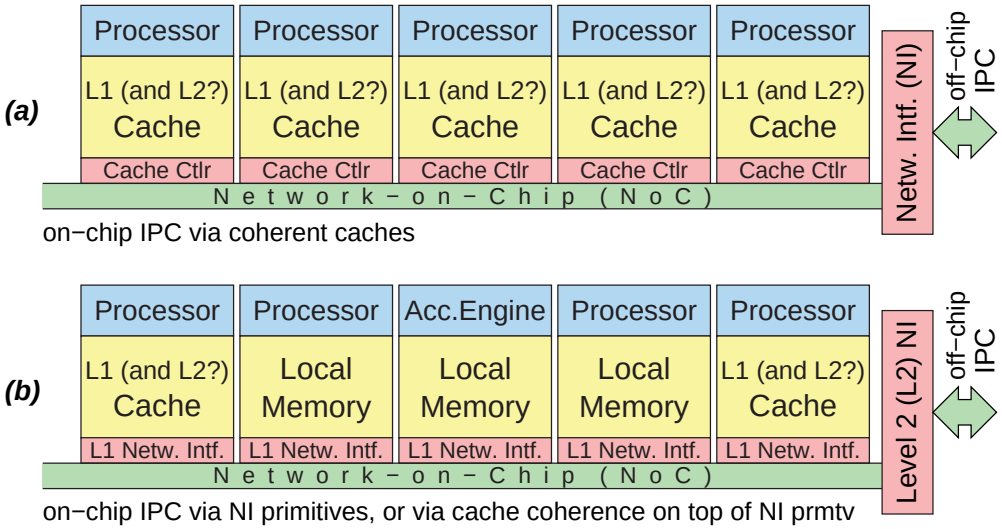

Fig. 2. NI placements for future systems: *(a)* NI only for off-chip IPC; *(b)* NI's for unified IPC.

1.3. Network Interface Placement in future CMP's

Figure 2 shows two alternative placements for network interfaces (NI) in future chip multiprocessors (CMP). The architecture in part *(a)* assumes that each processor has local caches (level-1, and possibly level-2 as well), and that all caches on a same chip are coherent. The coherence protocol runs on the network-on-chip (NoC); each node (processor plus caches) interfaces to the NoC via its cache controller. Communication among processors on a same chip is via coherent-cache shared memory. The network interface (NI), in its more-or-less traditional form, is used to communicate with the off-chip network; the NI is assumed to talk the same coherence protocol as the on-chip caches do [5], and processors communicate with the NI through this protocol: outgoing data depart from and incoming data arrive into the on-chip memory hierarchy. This architecture perpetuates the traditional dichotomy between local and remote IPC –the traditional view that NI operations incur a heavy overhead. In relative terms, the NI in this architecture is as "far" from individual processors as the NI in figure 1 was away from that processor.

Part *(b)* of figure 2 shows an architecture for unifying local and remote IPC –an architecture for bringing the NI much closer to each processor, so that *lightweight* (low-overhead) NI operations can be supported. The reasons for preferring this IPC architecture over the one of part *(a)* are as follows:

- Some CMP's, especially those intended for embedded applications, will contain processors that use local *scratchpad* –rather than cache– memory, or processors whose local SRAM can be configured to operate (partly) as scratchpad and/or (partly) as cache memory, e.g. to provide deterministic performance; the Cell processor [1] is one such example. These processors need a network interface –rather than or in addition to a cache controller– in order to communicate with the rest of the system. A similar situation occurs when processors are to communicate with special-purpose accelerator engines, as illustrated in the middle of figure 2*(b)*.

- For processors that use caches, or whose local SRAM can be configured to (partly) operate as cache, we propose to *merge the cache controller and the network interface* [7], since these have complementary functionalities. The basic function of the NI is to provide *data transport*, in the form of block transfers (section 2). Cache controllers have a higher-level function: handle cache misses and coherence events, using block transfers as an underlying primitive. This approach also

allows the option to implement some layers of the cache coherence protocols in software, using the underlying data transport primitives provided by the NI hardware.

- When network interfaces are provided *both* next to each processor (level-1 NI) and at the chip boundary (level-2 NI), the IPC protocols of large-scale systems can be *seamlessly extended* to the case where the communicating processors happen to reside on the same chip. These protocols can now feature low-overhead and low-latency, owing to the tight coupling of L1 NI's to the processors. At the same time, the high cost of off-chip communication (increased round-trip times, hence increased buffering) needs only be borne by the L2 NI's.

2. REMOTE DMA: THE GENERALIZATION OF LOAD/STORE INSTRUCTIONS

Communication occurs by transferring (copying) data from a source to a destination. Each transfer operation may concern various amounts of data: a single bit (rarely), or a single byte (inefficient, but used for text of media types), or a few bytes (one word, as with load and store instructions), or tens of bytes (e.g. a cache line), or hundreds of bytes (a small data structure?), or thousands of bytes (e.g. a small page), or millions or more of bytes (e.g. a large page or file). Hardware architects and application developers choose this granularity based on a well-known tradeoff: fine granularity for improved flexibility and space economy, versus coarse granularity for better amortization of operation overhead over a larger amount of work.

2.1. Granularity of Data Transfers as a function of Distance

Within the processor, the overhead of each transfer is the use of one word-wide path for one clock cycle; hence, transfers are performed at the word granularity. Between the processor and its local memory (e.g. level-1 cache, when caches are used), the transfer overhead is again the use of one word-wide path for one clock cycle, plus the consumption of one (word-wide) load or store instruction, plus the transmission, translation (TLB), and checking (against cache tags) of the (word-wide) address. In this case, single-word granularity is again used, for simplicity and flexibility reasons, although the overhead (on the order of two words –one instruction plus one address) could justify a granularity of a few data words per load/store instruction (and some processors do in fact provide such multi-word instructions). Between levels of the cache and memory hierarchy, each transfer incurs the cost of a few clock cycles in the cache controller, a bus transaction (a network-on-chip (NoC) packet, in future systems), and possibly a DRAM row activation (a few tens of nanoseconds). The transfer granularity, here, is the cache line –usually 32 to 128 bytes– which uses a typically 8-byte-wide datapath for 4 to 16 clock cycles; again, granularity is commensurate to overhead.

Besides the above, the only other type of data transfers supported by traditional computer systems were input/output (I/O) transfers, which occurred through the slow path discussed in section 1.2. Under these circumstances, the only other data transfers that traditional systems supported had to be on the granularity of tens of KBytes for them to achieve substantial efficiency.

In the future computing systems, where massive multiprocessing will be the default, all interprocessor communication, including to the storage and I/O devices, will be through the interconnection network (both on-chip and off-chip). The basic overhead of a data transfer through the interconnection network, in terms of buffer space and transmission capacity utilization, is the per-packet header, when compared to the packet's payload; these headers (plus CRC and other line overhead) are, today, on the order of 16 to 32 bytes, per packet. Other overheads, in terms of processing cost and complexity, are the per-packet routing, scheduling, and flow control decisions; there are known techniques, today, to limit all these to a few clock cycles per packet.

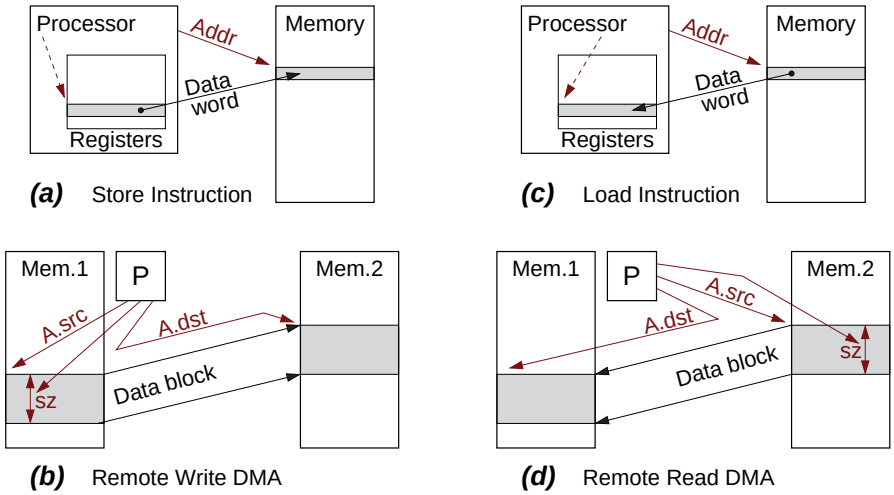

Fig. 3. Block data transfers (bottom) seen as generalizations of single-word instructions (top).

It follows from the above that well-designed, modern interconnection networks can provide good efficiency for data transfers at granularities as fine as 16 to 256 byte blocks: for packets of these sizes, header overhead is roughly between 6 and 50 percent, and typical packet time is 4 to 32 clock cycles, which can be overlapped with the per-packet routing, scheduling, and flow control processing. Designers must then strive to *also reduce all other IPC overheads* down to a single-digit number of (processor) clock cycles, so that interprocessor communication at all scales becomes efficient at granularities as fine as a few tens of bytes, i.e. one or more cache lines.

2.2. Remote DMA: Load/Store functionality at Block-Granularity

There is widespread agreement, nowadays, that *Remote Direct Memory Access (RDMA)* should be the basic hardware primitive for low-latency IPC, in multiprocessing environments other than cache-coherent MP's. Figure 3 supports this argument by pointing out that remote DMA is in fact the generalization of the load and store instructions on which processor-to-memory communication relies, when the transfer size grows from one word to an arbitrary size sz.

As illustrated in fig. 3(a), a store instruction specifies a source register number and a destination memory address, and causes a data copy from the former to the latter; the size of the copied data is implicit in the instruction opcode, and is typically one word. In an analogous way, a remote write DMA operation, fig. 3(b), specifies a source memory address, A_{src}, a destination memory address, A_{dst}, and a block size, and causes a copying of that block of data from the former to the latter address; in the general case, block size is arbitrary; we assume a global address space. Store instructions and write-RDMA's are "unidirectional" operations: control (address and op-request) and data move in the same, single direction. Load instructions, fig. 3(c), and remote read DMA's, fig. 3(d), are "round-trip" operations. This incurs a longer latency: first, control arguments (address and op-request) have to travel to the "remote" source, and then the response data (together with A_{dst}) travel back to the (local) destination.

As seen, the only fundamental difference between load/store instructions and remote DMA operations is the size of the data transfer concerned; differences in implementation tradeoffs result from this. Some researchers (e.g. [8]) have advocated very fast IPC using data transfers directly from or into processor registers. If register-based transfers were the only IPC mechanism, given the small size of the register file, the lack of an arbitrary-size transfer primitive would be a severe handicap. In a system that already

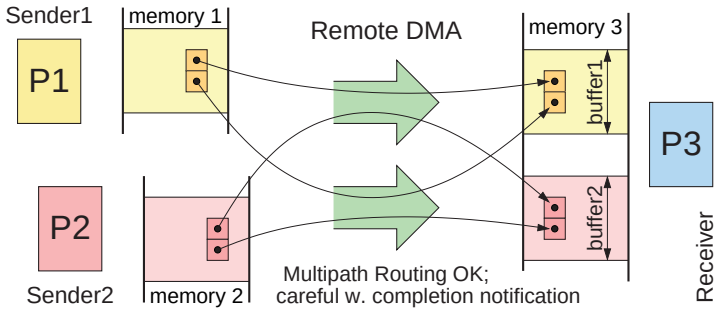

Fig. 4. Remote DMA requires single sender per receive buffer, or previous synchronization.

provides RDMA, we consider register-based transfers to only offer a very small added value: given all the other network overheads, the performance gain from saving a few register-memory transfers is quite limited. However, we do consider that RDMA initiation should be as fast as a few accesses to *local* memory (e.g. L1 cache). Remote DMA initiation requires two address parameters, A_{src} and A_{dst}. For the protection and migration support mechanism to properly operate, these arguments should be (gloabal) *virtual* addresses; this is further discussed in section 4.

2.3. Cache Protocols on top of RDMA?

In a multiprocessor with coherent caches, interprocessor communication is traditionally performed through the cache coherence protocol. Observe, however, that the handling of a cache read miss involves the equivalent of a remote read DMA for a block of size equal to one cache line. Also, a cache line write-back operation is the equivalent of a remote write DMA for a block of the same size. Thus, *RDMA-capable hardware may constitute an appropriate substrate for cache protocols to run on top of.* Section 3 (remote dequeue operation, and sections 3.4, 3.5, 3.6) discusses the possibility for network interfaces to trigger actions in response to events. Such a capability could be used to detect cache miss or cache coherence events and trigger corresponding cache line transfers via RDMA.

2.4. One-to-One Communication using Remote DMA

Figure 4 illustrates the RDMA operation, particularly in an environment where multiple parallel transfers exist, and the packets of each transfer may be routed through different paths ("adaptive" or "multipath" routing). 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 –otherwise the synchronization overhead between $P1$ and $P2$ would be excessive. Section 3.2 comments on what happens in situations where such separate buffer pre-allocation would be too expensive.

Multipath (adaptive) routing is desirable because it greatly improves network performance; however, multipath routing causes out-of-order delivery –a headache that many architects want to avoid. Remote DMA matches well with multipath routing: since each packet specifies its own destination address (and since destination regions are non-overlapping), it does *not matter* which packet arrives first and which one second –each packet goes and finds its own destination, and when all packets arrive the "puzzle" that they form will have been built. The only problem that remains is to detect when all packets belonging to a same DMA "session" have arrived. In systems guaranteeing in-order delivery, receiver software can detect that by polling a flag in the last byte of the DMA region. Systems that allow out-of-order delivery must provide other mechanisms to detect DMA completion, e.g. counting the number of bytes that have been received.

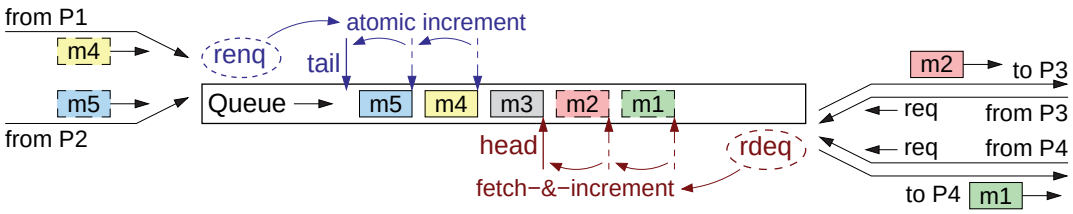

Fig. 5. Messages being atomically enqueued into and atomically dequeued from a Remote Queue.

3. REMOTE QUEUES: THE GENERALIZATION OF ATOMIC OPERATIONS

Individual load or store instructions do not suffice, as is well known, for synchronizing multiple threads running in parallel. Similarly, their block-granularity counterpart, remote DMA, does not suffice for synchronization. In shared-memory environments, a hardware primitive often provided for such purposes is the atomic test-and-set of a (single-bit) shared variable; in other cases, variations or slightly higher-level primitives are provided, e.g. (multi-bit) "fetch-and-op". We consider these hardware primitives to be of too low level, and inappropriate for scalable parallel processing. First, they carry a limited amount of information (one bit or one integer number), each. Second, the usual synchronization operations of parallel programing, when synthesized out of such primitives, require multiple round trips of control exchanges between the processors and the shared variables that are involved.

Optimized implementations of higher-level synchronization operations are provided by the MCS locks, which are based, to a large extent, on *shared queue* data structures. Similarly, *Remote Queues (RQ)* have been proposed as the basis for synchronization operations [9]. The author believes that remote queues are at the appropriate level of abstraction for them to constitute the basic *hardware primitive* for control and synchronization in scalable multiprocessing. Figure 5 shows the basic ideas of remote queues, and illustrates how these are generalizations of the atomic fetch-and-increment operations.

When multiple producers send information to a common location or consumer, the purpose of synchronization is to control the *interleaving* of data at the arrival site, disallowing meaningless intermixings. The *Remote Enqueue (renq) operation* does this for messages arriving from multiple sources. Messages $m4$ and $m5$, in figure 5, originate from different sources (or threads), $P1$ and $P2$, specify the *same address* as their destination, and happen to arrive simultaneously. Because their common destination address corresponds to a (remote) queue, rather than normal memory, the network interface *atomically* increments the queue's tail pointer and assigns separate, unique (normal-memory) write addresses to each of the arriving messages. We suggest to simplify the implementation by limiting the size of messages so that each of them fits into a single network packet; then, atomic message enqueueing becomes straightforward given atomic packet transmission over network links.

Dequeueing messages from (remote) queues is simple in the case of a single (local) receiver, but more difficult in the case of multiple (remote) readers. In the simple case, a single thread has read access to the queue, and the queue resides in its local memory –the queue is "remote" to the senders, but local to the receiver. In this case, atomicity of the dequeue operation is guaranteed by the single controlling thread. In a slightly more complex case, the hardware must ensure atomic dequeue's by each of *multiple* threads, still running on the *local* processor. These are equivalent to *atomic fetch-and-increment* operations: each thread atomically reads the queue head pointer –thus getting a pointer to the dequeued message– and increments that head pointer –so that no other thread can receive the same message.

The case of multiple *and remote* readers is more difficult. Its purpose is to serve in "job dispatch" type of applications: the queue holds descriptors for a number of pending jobs, and the readers are server processors; when a server becomes available, it dequeues a next job. Thus, dequeue operations must be

triggered by the (remote) readers and not by the queue itself. Figure 5 illustrates this by showing dequeue *requests* being sent by potential readers *at the time when* these readers wish to perform a dequeue. In the figure, requests from processors $P3$ and $P4$ happen to arrive at the same time; each of them, *atomically* dequeues a different (head) packet ($m1$ and $m2$ in the figure, in an unpredictable order).

Implementations of remote readers may vary. One alternative is to treat incoming request packets as *active messages* [10]: upon arrival, the local processor performs the dequeue operation and returns the dequeued message to the requestor. Another alternative is for the network interface *hardware* itself to interpret such request packets and perform the dequeue and dispatch operations [11]; this is analogous to having a finite-state machine (FSM) within the NI play the role of a (simple) processor running the active message code. We referred to such a NI capability as "triggering actions in response to events" in section 2.3, and we further use it in sections 3.4, 3.5, and 3.6. The next subsections describe or comment on several applications of (remote) queues in (asynchronous) I/O processing or in interprocessor communication and synchronization.

3.1. Queues as Generalization of Single-Item Communication Buffers

The most primitive form of producer-consumer communication is illustrated by the traditional teletype (tty) style I/O interface. Consider, for example, a keyboard-input interface consisting of one data register and one status register; the essential part of the status register is the *empty/full bit*, informing the receiver whether or not a new data byte has arrived in the data register since the previous one was read. Such a *single-item* interface imposes a tight interlocking of producer and consumer rates: if the consumer is too slow during even a short time period (a single byte-production period), a data-overrun error occurs. Conversely, if the consumer is too fast, it has to wait, using either polling (busy-wait) or an interrupt mechanism based on the empty/full bit.

Queues are the generalization of single-item interfaces; the larger they can grow, the more freedom they allow on producer-consumer rate-difference fluctuation. When there is a single producer and a single consumer process, it suffices to grow the *data* part of the interface to a multi-item form, e.g. a circular-buffer queue. The *control* part of the interface can still use a single-item form: a single head-tail pointer pair suffices. On the other hand, when there are multiple producers or multiple consumers, the control part of the interface must be generalized as well. The next subsection discusses the multiple-producer case; the multiple-consumer case is analogous, using the remote-dequeue operation, as discussed above.

3.2. Many-to-One Communication

Section 2.4 and figure 4 dealt with multiple producers sending data to a common consumer processor. In order for data transfers to proceed independent of each other (i.e. without needing prior synchronization), separate buffer areas must have been preallocated for each communicating pair. Each buffer area has to have its own control structure, e.g. a head-tail pointer pair, or empty/full bits per sub-block. There are two costs associated with such a setup. One overhead is for the receiver to poll the several buffer areas in order to discover where new data have arrived; we discuss this in section 3.4 below.

Another overhead occurs when the number of *potential* producers is much larger than the number of actual senders. In a system containing e.g. thousands of communicating processors (or threads), we may not know a priori which processor will want to send information to a given receiver P_r, and it would be too expensive to preallocate separate buffer areas at P_r for each and every other processor in the system. *Remote queues* (RQ) for *control* information is the proper solution in this case: a single RQ is set up at P_r for receiving communication requests by other processors. For every such request received, P_r allocates a dedicated buffer area, and responds with a pointer (in global address space) to this area. The requesting processor, then, uses remote DMA to put its data into this buffer.

3.3. Interrupts replaced by (Multi-Priority) Threads waiting on Queues or RDMA

An (I/O) interrupt is equivalent to a thread switch caused by a (presumably higher-priority) external event. In future multiprocessor environments (CMP or other), all I/O will be carried through network packets, similar to the other kinds of IPC. It follows that the only possible "external events" are the arrivals of particular kinds of network packets. Of particular interest are: *(i)* the arrivals of the last packet of a remote DMA "session", meaning that the receiving processor can now start processing the received data; and *(ii)* the arrivals of messages into previously-empty queues, where a thread was waiting to process messages from that queue. We assume the existence of hardware that switches among threads, especially when previously-blocked higher-priority threads now become ready to run. The author believes that interrupts should be replaced by such a mechanism, complemented by a method to register associations between NI events and processor threads.

3.4. Waiting for any of Multiple Events

There are cases where a thread wishes to wait for the occurrence of *any* one of a set of "interesting" events, as with the *select* system call in Unix, e.g. wait for any one of a set of (remote) queues to become non-empty, or wait for any one of a set of RDMA transfers to complete. This can be implemented in two ways. One alternative is for the NI to remember one thread ID per RQ: when an empty RQ becomes non-empty, that thread get unblocked. Multiple RQ's may specify the same thread ID to be notified; the thread should be informed about which RQ woke it up. A problem to be solved is how to inform the thread about multiple RQ's becoming non-empty, either before or while the thread starts processing the first (few) of them.

An alternative implementation can be through a network interface that can trigger actions in response to events, as discussed above on the occasion of the remote dequeue operation and in section 2.3. Consider a thread t_h that wishes to be notified when any RQ in a set S_h of RQ's becomes non-empty. First, allocate a "notification" queue Q_h for this purpose; make t_h wait on this *single* queue Q_h. Then, configure all queues in S_h so that, when they become non-empty, they each send a *notification message* to queue S_h. Using this method, thread t_h waits on a single queue Q_h, and finds in that queue a list of *all* currently non-empty queues in the set of interest, S_h.

3.5. Locks and Mutual Exclusion: Queueing up for Service

Traditional locks are used to control a set of processors (or threads) so that they can process a shared data structure in a mutually exclusive, i.e. non-overlapping in time, fashion. In a multiprocessor with remote queues, the equivalent functionality can be advantageously achieved as follows. The first case is when the processing to be performed, on behalf of each requesting processor, is less expensive than transferring the data structure to the requestor. In this case, it is preferable to perform all such processing *locally*, in a processor that happens to lie near the data structure (consider the future CMP systems as a "sea of memory" with processors interspersed in it). To make this work, set up a queue in the memory holding the data structure. Requesting processors remotely enqueue their request messages, containing the request arguments, into this queue. A thread on the local processor waits on this queue, and services arriving requests one at a time. Effectively, mutual exclusion is provided by request messages being atomically (remote) enqueued.

The second case is when transferring the data structure to the requesting processor is less expensive than the processing to be performed. In this case, it may be preferable to "ship" the data structure to one of the requestors, let that processor do its processing, have that processor ship the results back, then do similarly for another of the requestors, etc. Again, this would use a queue in the memory holding

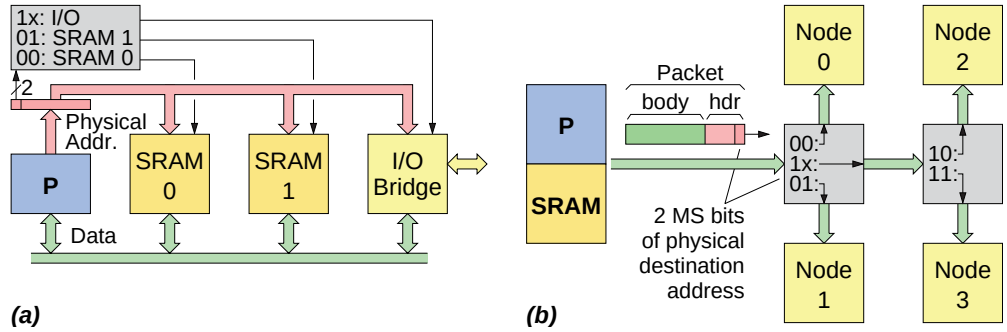

Fig. 6. *(a)* Physical address decoding in a uniprocessor; *(b)* geographical address routing in a MP.

the data structure; request messages are remotely enqueued into it. Either a local thread or the local NI interprets these request messages: for each of them, ship the data structure (e.g. via RDMA), then wait for a response (equivalent to releasing the lock), then service another request message, and so on.

3.6. Barrier Synchronization

A small-scale barrier synchronization can be implemented with one node (the "root") collecting phase-completion notifications from all participating nodes, and then in turn notifying all of them when that happens. A large-scale barrier should use a tree of collecting nodes and a tree of notifying nodes, in order to avoid the root becoming a communication bottleneck. Collecting and counting notifications –either at a tree node or at the root– can be conveniently done through a queue: each participant remotely enqueues a phase-completion notification. A computation thread, running on the processor that holds the queue, dequeues and counts the messages; when the prescribed count is reached, the thread sends a notification message to its parent node in the tree, or, if it is the root, to all its children. Alternatively, the barrier could be implemented completely in hardware, if the NI's can trigger actions in response to events, as discussed above: each queue may be set up so that it counts packet arrivals. In the present setup, an event should be signaled not for every packet arrival but when the number of arrivals reaches a prescribed count (just like a RDMA completion event occurs when the number of byte arrivals reaches a prescribed count). When this "event" occurs, the NI generates and sends one or more notification messages to other appropriate queues.

4. NETWORK ROUTING AS GENERALIZATION OF ADDRESS DECODING

Load and store instructions specify a memory address each, thus directing the data transfer to a specific memory word in a specific memory area. Figure 6*(a)* illustrates this "directioning" in a traditional uniprocessor, with a simple example of a memory system consisting of two SRAM chips (or blocks) and one (memory-mapped) I/O device. The most significant (MS) address bits are decoded and select a chip, then inside each chip a similar addressing tree directs the data transfer to a specific word in it. If we consider the data bus as a network connecting the processor to each word in memory, then the role of the memory address is to route the "access packet" to a specific destination reachable through that network. In this example, the address under consideration is the *physical* address of the load or store instruction, because it is the address that leads the transfer to a specific physical location in the memory or I/O circuitry.

Figure 6*(b)* illustrates the analogous situation for IPC –rather than load or store instructions– in a multiprocessor. A network packet, carrying part of a remote DMA or a remote enqueue message, specifies

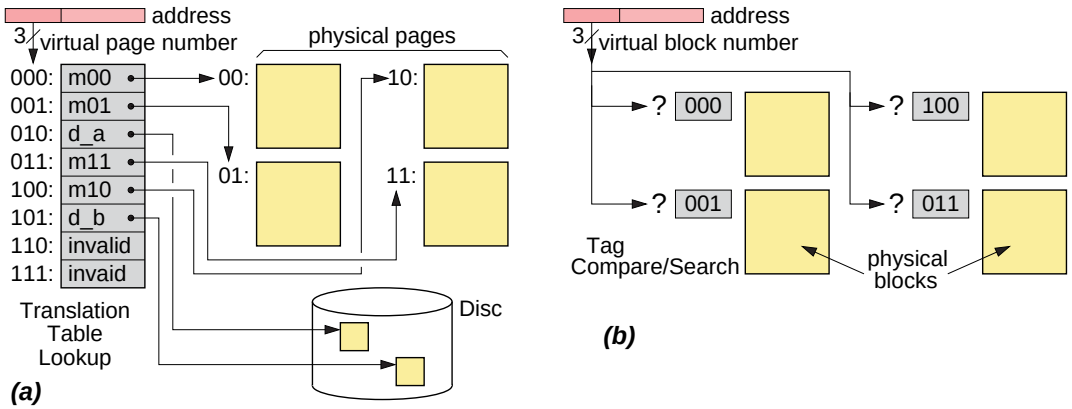

Fig. 7. Two methods to support data migration: *(a)* translation table; *(b)* cache style.

a destination address in its header. The network routes packets based on these destination addresses. Considering the total memory in the system as a "global memory space", *network routing* is clearly seen to play the same role in the multiprocessor as *address decoding* played in uniprocessors. In figure 6 addresses are *physical*: they correspond to the "geographical" placement of memory modules in the system, and this correspondence is "hardwired" in the routing function and cannot change at runtime. As we discuss immediately below, such a correspondence is too restrictive because it does not allow the operating or runtime system to *transparently* migrate data. *Virtual-to-physical address translation* is introduced for that purpose.

4.1. Address Translation in support of Transparent Data Migration

Performance critically depends on *locality*, and this dependence becomes stronger and stronger with successive technology generations –this is the famous "memory wall" problem. Data is continuously migrated in modern systems, in an attempt to improve locality: as the *working set* of the data that a processor accesses changes during the progress of application execution, "old" data are sent further from the processor and "new" data are brought closer to it.

Some application programs prefer to *explicitely control* data migration themselves: they specify the exact moments in time and addresses in space where data transfers should occur. Application developers do that –e.g. in the embedded domain– in order to achieve *predictability* in performance-critical applications that are well enough understood by the programmer for him or her to explicitely specify these transfers. Such applications that manage data migration themselves can operate with *physical* memory addresses, as above: when referring to a specific data structure, the application knows, at each point in time, which (physical) address to use, depending on where this structure was migrated to last time.

For the majority of the applications, though, data migration is performed *transparently* to them, by the operating or runtime system and by the hardware. Figure 7 illustrates the two methods to achieve that. *Address translation*, shown in part *(a)* of the figure, is used when the number of allowed locations, for a logical item, is large; the operating system uses this method, when migrating data ("paging") between main memory and disk. This method uses a list (translation table) of all logical items (virtual pages); for each of them, the list specifies its physical location. *Cache lookup*, shown in part *(b)* of figure 7, is the other method for transparent data migration; the hardware uses this method when migrating data between dynamic RAM and various SRAM blocks –some closer to the processor and some further away from it (the levels of the cache hierarchy, or other processors' caches). Cache lookup avoids the delay of

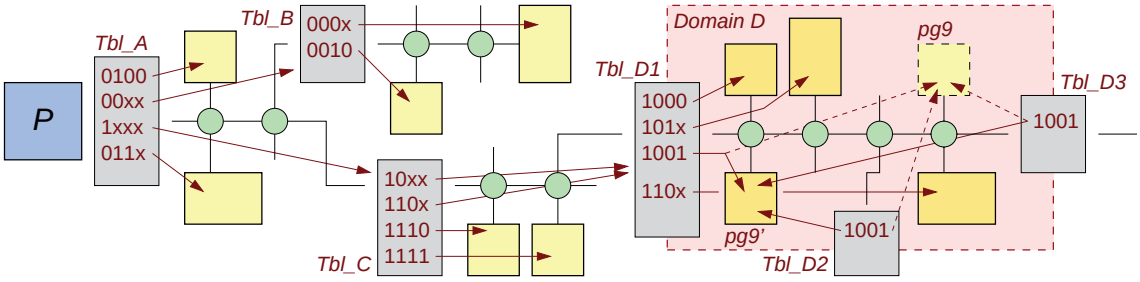

Fig. 8. Progressive translation/routing: packets carry virtual addresses, tables provide physical route (address) for the next few steps.

the translation table by restricting the number of locations that each logical item is allowed to migrate to (using set associativity); on the other hand, it introduces the cost of *searching* for the desired item by performing multiple tag comparisons –often in parallel, consuming energy. Cache lookup is *not* scalable to cases where data are allowed to migrate far away: it requires a broadcast of the search tag and multiple comparisons.

4.2. Progressive Translation for Localized Updates upon Migration

Old uniprocessors performed virtual-to-physical address translation at only one place –the processor's TLB. Newer systems need multiple TLB's –one per processor, plus a TLB in the network interface, in order to provide user-level access to the NI: the user must provide *virtual* addresses as arguments, and the NI has to perform the (protected) translation to physical addresses for the operation to be executed. When a data item (e.g. a page) migrates, *all* copies of the translation table or portions thereof (e.g. TLB's) need to be updated, which is a problem as this number increases. For scalable multiprocessing, this system has to be revised: we cannot afford all virtual-to-physical address translation to occur at the source node of an IPC packet, because that would mean that all nodes (may) know the current (physical) location of all (virtual) data objects, hence all nodes may have to be notified when an object is migrated.

Figure 8 illustrates a multi-level translation scheme that can solve this problem; let us call it *progressive translation*. Each packet (imagine that it starts from processor P) may pass through several translation tables on its way to its destination. The packet *always* carries its *virtual* destination address. Each translation (routing) table directs the packet for a few more steps (through a few more switches –the round devices in the figure) towards its final destination. Each table contains accurate (definitive) location (routing) information for the data items (e.g. pages) that are close to it, but only approximate information for those that are further away: "go to that table and ask there", it says....

Imagine, in figure 8, that (virtual) page 1001 migrates from (physical) page-frame $pg9$ to frame $pg9'$, and that, while doing so, it stays within the same *domain* (neighborhood) D. Then, only the translation/routing tables at the entries of and inside D need to be updated –assuming that all external tables merely point to an entry point into D, rather than specifying exact location within D.

Figure 8 also illustrates a couple of practical aspects of progressive translation. The function of the tables can be interpreted to be to provide *run-time configurable* network routing, rather than fixed "geographic" routing: each destination address is allowed to change its location in the net. However, it would be too expensive, in general, to provide this capability inside each network switch –especially so inside the switches of networks-on-chip (NoC). The figure shows individual switches as little circles, and shows translation/routing tables positioned every few switches –but not in every single switch. Each table provides *physical* route information through a sub-net –but not through the entire, system-wide network.

For example, in a CMP, each processor may have a TLB containing information only about on-chip data, while off-chip data are only described in table(s) residing at the off-chip interface(s). This placement of tables as interfaces between sub-networks is similar to the "routing filters" in the "Wormhole IP over ATM" proposal [12].

In a large multiprocessor system, with the many millions of objects in its global address space, translation/routing tables would be hopelessly large, if we could not aggregate together entries referring to objects that are nearby in both logical and physical space. *Variable-size pages* allow this in a convenient way –figure 8 assumes this by including "don't care" bits (marked x) in page numbers. An implementation problem (cost) will arise when a small portion of a larger space migrates to a physical area outside the location of the large space –e.g., in the figure, if page 1001 were to migrate to the far left, outside the region seen as 1xxx by processor P. Internet routing tables handle such situations using *longest-prefix matchings*: when the table contains both an entry for 1xxx and 100x, addresses 1000 and 1001 are translated using the latter (longest) entry that they match, 100x, while addresses 1010, 1011, 1100, ..., 1111 are translated using the former entry. Searching through such tables at high speed is expensive, but at least the problem has been researched extensively [13].

4.3. Protection, seen as a case of Firewalling

Besides migration support, the other fundamental reason to use virtual addresses is *protection*: not all processes are allowed to generate all addresses hence access all data. In a system like the one of figure 8, the first line of protection is at the processor generating the IPC operation: the operating system must ensure that the process has access permission to the virtual addresses that it specifies. In large systems, it is further desirable to be able to *partition* the machine so that one partition does not even trust the operating system of the other. In figure 8 this can be achieved by having the translation/routing tables at the entry points of a partition –e.g. *Tbl_D1, Tbl_D2, Tbl_D3* for partition D– operate like Internet *firewalls*: have a list of critical data objects (address regions) in partition D that processes running outside D are not allowed to access, hence filter out and drop all such "illegal" packets entering D from the outside.

4.4. Where to keep Flow and Congestion Control State

One of the hardest problems in interconnection networks is flow –and especially congestion– control. Effective solutions to these control problems generally need to keep per-destination state. In a system like the one of figure 8, the translation/routing tables only seem as a natural place for maintaining such state, since their entries are related to geographical locations through the net. The fact that single entries may correspond to very large subnetworks that are very far away may not be a problem, but rather a challenge for *hierarchical flow/congestion control*.

5. CONCLUSION

Interprocessor communication (IPC) is increasingly important in the future systems with increasing numbers of (cooperating) processors. IPC is a case of locating and transporting data. Data transport is the generalization of load and store instructions from one word to arbitrary sizes: *remote DMA*. Synchronization (coordination) of communicating processes can be advantageously done through *(remote, multi-access) queues*. Identifying and locating data objects –i.e. addressing– should be done so that migration and protection are supported. In this respect, large-scale multiprocessors may have to adopt some of the techniques used in Internet routers.

6. ACKNOWLEDGMENTS

The viewpoint and opinions expressed in this paper have been reached by the author following the discussions and the work performed within the "Scalable Computer Architecture (SARC)" integrated project #27648 of FP6, supported by the European Commission. These discussions have been led and steered by Stamatis Vassiliadis; the project itself owes its existence to Stamatis Vassiliadis. Stamati, *thank you!* –I miss you; our project and our scientific community misses you; Samos is missing you. It is too bad that you departed so soon –you could have helped us much more, we would have enjoyed your company much more. Καλό ταξίδι αγαπημένε συμπατριώτη, συνεργάτη, φίλε –farewell dearest compatriot, colleague, friend; we will remember you forever!

Other colleagues who helped formulate these views include Stamatis Kavadias, Georgi Gaydadjiev, Michael Papamichael, Angelos Bilas, Babak Falsafi, Christos Sotiriou, Spyros Lyberis, Vasilis Papaefstathiou, George Kalokerinos, Manolis Marazakis, Angelos Ioannou, Jose Duato, and Ian Johnson. Many thanks to all of them!

7. REFERENCES

[1] 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.

[2] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded SPARC processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar. 2005.

[3] Intel, "World's first quad-core processors for desktop and mainstream servers," in *http://www.intel.com/quad-core/*.

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

[5] 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.

[6] 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.

[7] G. Gaydadjiev, "Personnal communication," 2006.

[8] S. Keckler, W. Dally, D. Maskit, N. Carter, A. Chang, and W. Lee, "Exploiting fine-grain thread level parallelism on the MIT Multi-ALU processor," in *Proc. 25th Int. Symposium on Computer Architecture (ISCA'98)*, June 1998, pp. 306–317.

[9] 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*, June 1995, pp. 42–53.

[10] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Proc. 19th Int. Symposium on Computer Architecture (ISCA'92), Gold Coast, Australia*, May 1992, pp. 256–266.

[11] S. Kavadias, "Network interface support for synchronization primitives," in *HiPEAC ACACES summer school Poster Session, L'Aquilla, Italy*, July 2006.

[12] M. Katevenis, I. Mavroidis, G. Sapountzis, E. Kalyvianaki, I. Mavroidis, and G. Glykopoulos, "Wormhole IP over (connectionless) ATM," *IEEE/ACM Trans. Networking*, vol. 9, no. 5, pp. 650–661, Oct. 2001.

[13] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network*, vol. 15, no. 2, pp. 8–23, Mar. 2001.