Hardware Support for Explicit Communication in Scalable CMP's

Carlos Villavieja[1], Manolis Katevenis[2], Nacho Navarro[1], Dionisios Pnevmatikatos[3], Alex Ramirez[1], Stamatis Kavadias[2], Vassilis Papaefstathiou[2], and Dimitrios S. Nikolopoulos[2] ⋆

[1] Barcelona SuperComputing Center, Universitat Politècnica de Catalunya
[2,3] Foundation for Research & Technology - Hellas (FORTH), Institute of Computer Science (FORTH-ICS), Heraklion, Crete, Greece

- members of HiPEAC -

Abstract. Programming models with explicit communication between parallel tasks allow the runtime system to schedule task execution and data transfers ahead of time. Explicit communication is not limited to message passing and streaming applications: recent proposals in parallel programming allow such explicit communication in other task-based scenarios too. Scheduling of data transfers allows the overlap of computation and communication, and latency hiding, and locality optimization, using programmable data transfer engines, such as prefetchers or DMA controllers.

In this paper, we present a qualitative analysis by comparing explicit communication scenarios in two different shared memory CMP architectures. The baseline architecture uses caches, a directory-based coherence protocol, and programmable prefetchers for scheduled data transfers. The target architecture uses on-chip local memories that are globally accessible with regular load / store instructions, and programmable DMA controllers for scheduled data transfers. The local memories architecture scales better, because it does not require a coherence protocol.

Our analysis shows that the use of on-chip local memories, and remote-read / remote-write operations reduces access latency to critical data, network traffic, and energy consumption.

1 Introduction: Explicit Communication

Parallel processing is necessary to take advantage of current and future multiprocessor systems. Parallel processing consists of computation and communication. Performance of computation critically depends on locality: data being

⋆ [2] also with the Univ. of Crete, Heraklion

⋆ [3] also with the Technical Univ. of Crete, Chania

⋆ This work has been supported by the Ministry of Science and Technology of Spain and by the European Union (FEDER) under contract TIN2007-60625, the HiPEAC European Network of Excellence (IST-004408) and the SARC European Project(EU contract 27648-FP6). All products or company names mentioned herein are the trademarks of registered trademarks of their respective owners.

close to the core that processes them. Communication consists of data and synchronization moving from producing to consuming processors. In all cases, *data placement and movement* is critical.

Programming productivity has dictated a preference towards *shared memory* models of parallel programming, often implemented on top of *coherent caches*. These lead to data placement and movement occurring mostly under hardware control, with simple or simplistic algorithms, and with little opportunity for the programmer or runtime software to orchestrate them. Explicit Communication allows the runtime to control data placement and movement better.

Explicit Communication is the basis for parallel programming models like message passing or data streaming, which expose the data communication among the different parallel tasks. In shared-memory programming models, recent proposals have introduced the notion of task-driven parallelism, where the programmer identifies potentially parallel tasks, and their inputs and outputs. From that information, the runtime system can detect the same dataflow information as provided by message passing or streaming.

Shared-memory programming models, such as OpenMP [17], UPC [15], and CoArray Fortran [3], assume implicit communication through loads and stores. It has been shown that implicit communication over a coherent shared address space outperforms explicit communication in applications with *irregular* data access patterns, where it is hard to distribute or prefetch data optimally in local memories [4, 23]. On the other hand, in applications with *regular* data access patterns, implicit communication often incurs unnecessary coherence-related traffic.

The advent of multi-core processors with software-managed local memories, such as GPUs and the Cell/BE, has stimulated the introduction of programming models with explicit communication and explicit identification of task working sets. Programming models such as RapidMind [18], Sequoia [7] and CellSs [1], use attributes to identify the input and output data of parallel tasks, thereby enabling the runtime system to prefetch working sets to the local memories of processors. Explicit communication can be *integrated* with shared-memory programming models such as OpenMP [5, 6, 8], to enable better locality management while preserving the abstraction of a shared address space.

As an example of explicit communication, Figure 1 shows a matrix to be processed in blocks, and the inter-block dependency pattern. Processing each block is a potentially parallel task, that reads data from its two neighboring blocks (above and to the left). Once all the dark grey blocks have been processed, blocks 1, 2, and 3 are ready for parallel execution. The runtime system detects that after the latter complete, blocks A, B, and C will be ready for parallel execution, and so it can schedule their execution and the corresponding data transfers ahead of time.

Implicit Communication, as opposed to explicit, happens when the data to be consumed by a task is not known before task execution –for example if we do not know which blocks in the matrix will be required by a task.

This paper presents a qualitative analysis of how two different CMP architectures support explicit communication in parallel applications. The first archi-

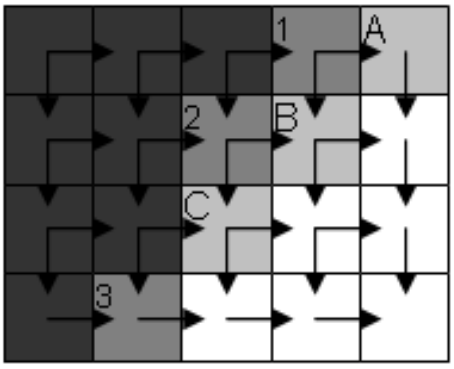


Fig. 1: Frame matrix of a computed image

tecture, "C+PP", uses *caches*; support for explicit communication is provided by *programmable prefetchers*. We assume a multi-level cache hierarchy and a network-on-chip (NoC), with local caches close to the processors and a global shared cache. This architecture relies on a directory-based hardware protocol to maintain coherence among the copies of the same data in the different caches. The second architecture, "LM+DMA", uses globally accessible on-chip memories, located close to the processors, instead of caches. Explicit communication is supported by these *local memories*(LM) being accessible via load / store instructions from any of the processors, as well as via remote direct memory access (RDMA); thus, they actually behave as an on-chip NUMA system. Since there can be only one physical location for a particular piece of data, there is no need for a coherence protocol.

Our analysis shows that the architecture using local memories requires much less network traffic in order to perform the required data transfers, compared to the cache architecture. This NoC traffic reduction not only has a potential performance impact, but also makes the architecture more scalable and less power-hungry.

2 Related Work

Several recent studies looked at on-chip memories and prefetching techniques to improve application performance, exploiting the reduced latency and the high bandwidth that are available on-chip. At the same time, several programming models are being introduced to use these features.

Leverich et al. [14] present a comparison of two memory models, cache and local memories, like ours; they compare performance and energy in streaming applications, using a simulator. They conclude that there are cases where either system has advantages, but, unlike us, they generally found little advantage in having local memories, hence they conclude that local memories are not worth

their added cost. Our conclusions differ from their's, for the following reasons: *(i)* their applications were mostly compute-bound, hence communication performance had a reduced impact; *(ii)* their applications had very little *on-chip* communication, among the cores –most communication was between each core and the off-chip main memory; *(iii)* they studied small-scale CMP's, where coherence works via a shared bus, while we study scalable CMP's that use a NoC and directory-based coherence; they do comment that larger CMP's are likely to benefit more from the "streaming" communication style.

Gummaraju et al. [11, 12], study how to map stream programs into General Purpose Processors (GPP). The streaming programming model is demonstrated to be very helpful to hide memory latencies. They present some hardware modifications to single core GPP. They introduce L2 cache lines blocking mechanism and a programmable software prefetcher. These modifications allow to use partially an L2 cache as a Local Memory. They finally build a hybrid Cache/Local Memory to mitigate memory latencies. This paper complements Gummaraju's work. We have studied several communication patterns which can be applied to [11] and many future CMP architectures.

Several producer-initiated mechanisms, including update-based coherence and locks[10], have been evaluated in [2] for distributed memory coherent-cache based systems. Prefetching is shown to perform well on average, while for predictable workloads the best-performer is Streamline, which is an L2 cache-based message passing mechanism. RDMA had not been evaluated in that study, like we do here.

Rangan et al. [21] consider high frequency streaming with queues between producers and consumers. They propose per entry full flags in the same cache-line with the data, which allow C+PP to avoid fence operations or other synchronization. This approach combines synchronization with the data transfer and should allow hiding all the latency of global communication either with prefetching or with update operations, but would require more network packets and hence more energy than LM + DMA. They evaluate write-forwarding (similar to update) and two other hardware-aggressive designs. Their baseline system provides a very fast bus; traffic or energy consumption is not reported. They also argue that pairwise nature of inter-thread interaction allows insight for large-scale CMPs of the future, from their two-core setup.

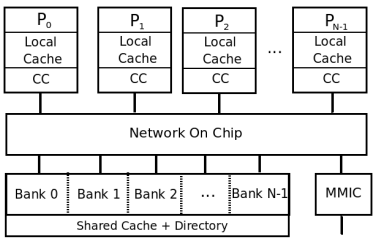
Mukherjee et al. [20] use a queue in memory to communicate from and to a coherent network interface. Synchronization is done with a lazily updated shadow head pointer for the producer (which requires fence operations) and per cache line valid bits with sense reverse for consumer synchronization. Their approach exploits a network interface that modifies the behavior of the coherence protocol for queue access and exceptionally treats as an enqueue/dequeue the request for a subsequent cache line (next queue entry). This optimization would require modification of the directory controller and, as Rangan et al. [21], requires cache-line sized buffers.

Both studies above did not consider DMA, or corresponding hardware techniques to improve data copy in batch.

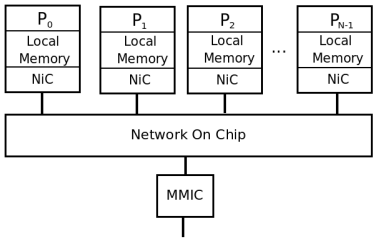
3 Architectures under Comparison

In this section we describe the two on-chip memory architectures, cache and local memory, compared in this paper, in the context of a scalable chip multiprocessor (CMP). The CMP is composed by multiple cores, on-chip memories, and a Network on Chip (NoC). The system architecture is based on a single global address space, and all memory locations are accessible by all processors via load/store instructions, that can access either local or remote memory locations. The translated physical addresses distinguish between local and remote accesses. The two memory architectures that we study are: *(i)* Cache-based architecture with directory-based coherence, extended with programmable prefetchers, called *C+PP* and shown in Figure 2a (CC stands for cache controller); and *(ii)* Local Memory based architecture, with programmable DMA controllers, without caches, called *LM+DMA* and shown in Figure 2b (NiC stands for network interface controller). In both architectures, MMIC stands for main memory interface controller.

Both architectures offer a flat memory view and are able to run shared memory applications, with some differences. In the cache case (C+PP), the cache coherence protocol makes sure the multiple copies are coherent, and data replication and migration is implicit. Performance can be improved using programmable prefetchers. In the local memories case (LM+DMA), no coherence protocol is implemented for the local memories, therefore data replication, migration and consistency need to be explicitly performed by software, through DMAs and remote read/write operations. Emergent programming models that explicitly manage memory hierarchies [1, 7, 18] leverage compiler and runtime system support to automate DMA transfers, along with reads and writes to and from local memories.



(a) Cache-coherent architecture with programmable prefetcher (C+PP)



(b) Local Memory based architecture with DMA (LM+DMA)

Fig. 2: Architectures compared

3.1 Cache Architecture and Communication Mechanisms

The C+PP architecture of figure 2a features a local cache per processor. Each cache controller (CC) contains a programmable prefetcher. The CMP also includes an on-chip shared (L2) banked cache. All caches are kept coherent using a directory-based protocol; the directory is kept in the shared cache. C+PP supports the following cache-based communication mechanisms, with optional variations:

- *Read Miss:* the basic communication mechanism to request the result of a previous computation that resides somewhere in the system; responsibility to locate that datum is left to the directory hardware.
- *Prefetch:* analogous to mass read-misses; fetches one or more lines into the local cache.
- *Flush:* evict data from the local cache, sending them to the shared cache; notice that data are sent to a lower cache level, and not to another node's local cache.
- *Write-Update* (found in some systems, possibly applied to only a subset of all data): on write-hit, send the written data to remote cache(s) that already have copies of the written address, rather than invalidating their copies [9] [19].
- Write-Miss variations-optimizations: *No-fetch on write-miss*: do not fetch the old contents of the cache line on write-miss; requires per-word validity flags. *No-allocate on write-miss*: do not allocate the line being written, but rather collect write-data in a write-combining buffer [13], and send them to be written elsewhere, as for flush or write-update.

3.2 Local Memory Architecture and Communication Mechanisms

The LM+DMA architecture of figure 2b has a local memory per processor. All processors include a Network interface Controller (NiC) that features a programmable DMA engine to transfer data between local and remote memory (being either the LM of another processor or off-chip). Since there is a single global address space, any processor can access any LM in the CMP. LM+DMA supports the following communication mechanisms:

- *Remote Read (Load)*: load instruction, from a remote node's LM or from main memory; analogous to a read miss, except that it leaves no local copy around.
- *Remote Read DMA*: send a request to another node's NiC to perform a remote-write DMA, transferring a block of data to this node's local memory; analogous to prefetch, except that the local copy and the original are at different addresses, and will *not* be kept coherent with each other.
- *Remote Write (Store)*: store instruction into a remote node's LM or main memory; most effective with a write-combining buffer [13]; analogous to a write-miss with no-allocate and with write-update.

– *Remote Write DMA*: copy a block of data from local memory to some other node's LM or to main memory; analogous to flush with write-update (a rarely encountered combination). DMA variations-optimizations: strided or list-based scatter/gather DMA.

Note that this architecture offers a single memory level compared to C+PP that also offers an L2 cache. For a fair comparison, the memory sizes in the two architectures should be similar. This can be achieved either by increasing the size of the local Memories in LM+DMA, or by adding a second level of Local Memory, shared between the local nodes. We do not consider these options in this paper as they would complicate the architecture comparison.

4 Communication Patterns and Mechanisms

Communication occurs every time a thread reads a word that has last been modified by another thread. We call *producer* the modifying thread, and *consumer* the reading thread. Although the terms producer and consumer have been traditionally associated with stream processing, the above definition clearly indicates that these are completely general and apply to all cases of shared-memory programming. Performance and cost are critically affected by the patterns in which communication is expressed in software and the mechanisms by which hardware implements it. This section categorizes and compares communication patterns; mechanisms were listed in the previous section, and are briefly commented here.

4.1 Push or Pull, per-Word or Batch Communication

A consumer gets paired up to a producer either directly or indirectly, through their common use of a same variable address. Between production time, when the word was last modified, and consumption time, when the word is read, the (new) data must physically travel from producer to consumer. This transfer of data can occur in one of various communication patterns, depending on the combination of application code, runtime libraries, and hardware architecture. These patterns consist of choices made in four issues: *(i) initiator and time* of transfer - at production time, consumption time, or scheduled in-between; *(ii) location* where data are placed - at producer, consumer, or some "central" location; *(iii) data transfer unit* - words, cache lines, or larger blocks; and *(iv) names* (addresses) of the data copies involved. Figure 3 illustrates the most usual communication patterns. In this figure, time is drawn horizontally, and space is shown vertically. Notice that in well-written parallel programs, some form of synchronization will occur between production and consumption time (dash arrow).

The first two cases, (a) and (b), in fig. 3 are *pull* type communication, where data transfer is initiated by the consumer. Case (a) is the default shared-memory communication, without any prefetch or other optimization. The producer updates its local copy; if this is a cache and the consumer had an old copy, the latter will get invalidated (not shown). When the consumer later discovers that

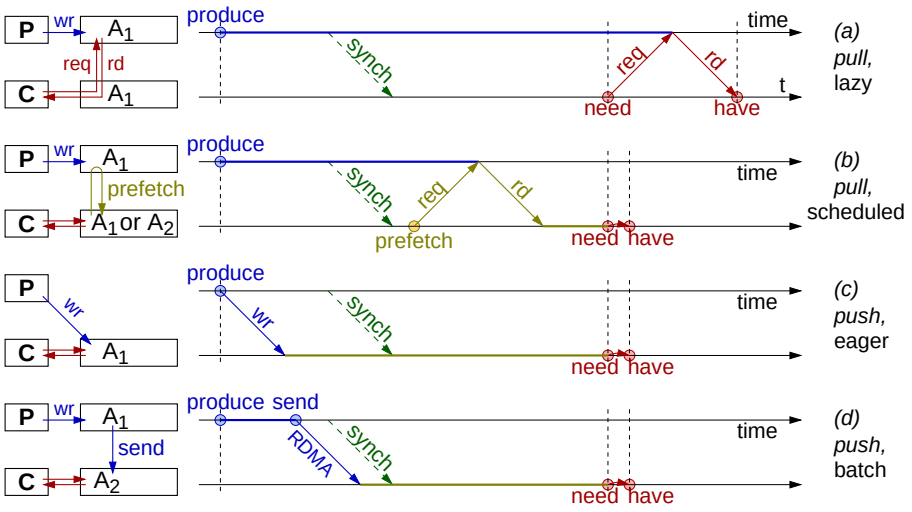


Fig. 3: Communication patterns in space and time.

it needs the (new) data, it suffers the latency of a miss or a remote read. Case (b) is the *prefetch* optimization: the consumer initiates the transfer *before* the data are needed, thus hiding the fetch latency by overlapping it with other useful computation. For effective latency hiding, prefetches of input data should be overlapped, i.e. multiple transfers must be ongoing in parallel. Explicit communication programming allows the runtime library to batch and schedule these prefetches of the entire input data set or a large portion of it. In the cache case, the prefetched copy has the same address, A_1, as the original data. In the local memory case, prefetching is via remote read DMA, which creates a copy with a different address, A_2.

Cases (c) and (d) of fig. 3 are *push* type communication, where data transfer is initiated by the producer. Push transfers are generally more efficient than pull ones, because push avoids the overhead and latency of the request packet; the difficulty is in the producer knowing its one or many consumer(s) at production time. A common case where the runtime environment can benefit from push type communication is parallel task creation or triggering, where a producer typically sends task descriptors and data to many consumers for execution at once. Case (c) illustrates *remote write*: the producer writes directly into consumer's local memory [16]. With caches, this can be achieved using *write-update* [9] –instead of the usual write-invalidate– protocols; note that for the update to reach the consumer's private cache, that cache must have had copies of the concerned words from beforehand, otherwise the update only reaches the shared cache. In our assumed LM+DMA architecture, store instructions are allowed to use addresses of other processors' local memories, resulting in directly writing into a remote memory. In both cases, NoC packets carrying single-word remote writes

or updates are inefficient (large header overhead compared to payload data); for efficiency, one needs write-combining buffer(s) [13], and the application to only write once to each target address and the address order to be "convenient". As illustrated in the figure, remote write economizes in memory utilization, because it does *not* consume any space in the local memory or cache.

Case (d) is a variation of (c), which can remedy the inefficiencies –if they exist– resulting from skewed write-address order, or multiple writes into a same address, or lack of write-combining buffer: updates are first collected in local memory, and then sent in batch(es) to the consumer(s). Such batch or block sending is typically done via remote write DMA, as illustrated in the figure with the distinct addresses A_1 and A_2. The equivalent in cache-based systems would be similar to a dirty-line flush operation (with write-update to remote private caches) that one rarely (if ever) finds in any real system. Batch sending of a block via RDMA is efficient when most words (or most cache lines) in the block have been updated –otherwise, one would prefer only the updated words (or cache lines) to be sent at write time or to be requested at read time. In the case of sparse updates, *gather*-type RDMA can improve efficiency. Overall, case (d) is the producer-initiated counterpart of case (b) –the producer-initiated prefetch; (d) can be started earlier than (b), before synchronization, and avoids the read/prefetch request overhead.

4.2 Underlying, Differentiating Traits

The two architectures under comparison differ in three fundamental architectural characteristics; these are the reasons underlying the differences in efficiency and performance to be seen in the next section:

Migration Support. Caches assume that addressable objects move (migrate) very frequently, in fine granularities (cache lines), and that software is *not* able to keep track of where each such object currently resides. Hence, to locate an object, given its address, hardware support is needed. Small systems locate objects by tag snooping; large systems locate objects using a centralized directory. The overheads associated with the coherence directory are the major cost factor differentiating C+PP from LM+DMA, as we will see in the next section. *Local memory* systems, on the other hand, assume that (runtime) software is able to keep track of where each object currently resides. Hence, there is no reason to maintain this information in a hardware directory, nor to consult that for each and every individual cache-line worth of data.

Addressability of Local Memories. The local memory of a node is directly addressable, by either the node itself or by any other node. By contrast, a node's cache locations do *not* have addresses of their own. Thus, push-type communication is straightforward with local memories, but awkward or impossible with caches –caches are obliged to work mostly or only with pull-type communication.

Coherence upon Copying. Both architectures improve read latency by making local copies from remote originals. Caches, in addition, consider themselves obliged to maintain all copies coherent to their originals, forever thereafter; this

is good if and when software really needs it, but it is useless overhead when not needed (e.g. after recycling a communication buffer).

5 Use Cases

Several use cases occur, depending on the input and output data location of a certain task. We study these cases, taking also into account on-chip/off-chip location. For each case, we describe the network packets required to transfer a block of data of size N cache lines; for DMA we assume that each network packet carries one-cache-line worth of data. In table 1, all memory operations are evaluated counting the number of network packets. Table 1 is divided in two vertical groups, fetch and write operations. For each operation, the two studied architectures are shown and for each one, the location of (input/output) data is considered, off-chip or remote memory(rmem) on-chip. All possible operations/locations are detailed in the first column, by the packet type. In the last two rows, the total number of packet is shown. Some operations show different number of packets depending on the type of communication. For all packets in the table, a subindex shows the order of packet communication in the NoC.We have omitted the time to program the prefetcher or the DMA controller, assuming that these are equal in the two architectures.

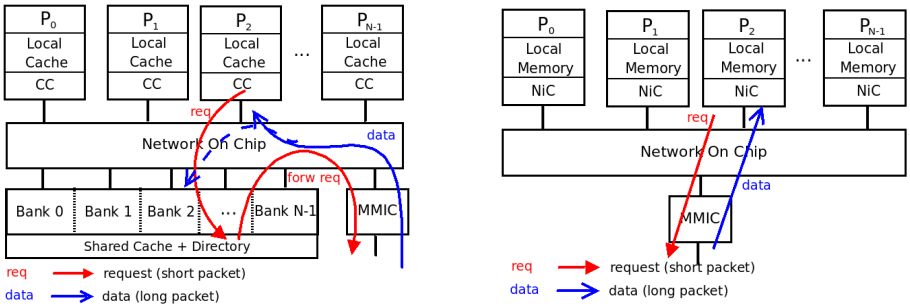
	Num.Packets							
	Fetch				**Write**			
	Cache	**LM**	**Cache**	**LM**	**Cache**	**LM**	**Cache**	**LM**
Packet type	**off-chip**	**off-chip**	**rmem**	**rmem**	**off-chip**	**off-chip**	**rmem**	**rmem**
(Pre)fetch Dir req	$N(on)_1$	-	$N(on)_1$	-	$N(on)_1$	-	$N(on)_3$	-
MM request	$N(on)_2$	-	-	-	-	-	-	-
Dir forw req	-	-	$N(on)_2$	-	-	-	$N(on)_{4,2}$	-
Dir miss-MM req	$N(off)_3$	-	-	-	-	-	-	-
MM reply	$N(off)_4$	-	-	-	-	-	-	-
Dir inclusion	$N(on)_5$	-	-	-	-	-	-	-
Reply to prefetch	$N(on)_6$	-	-	-	-	-	-	-
Rmem reply	-	-	$N(on)_3$	-	-	-	$N(on)_5$	-
Inv to Dir	-	-	-	-	-	-	$N(on)_1$	-
Dir inv forw	-	-	-	-	-	-	$N(on)_2$	-
DMA program	-	-	-	$1(on)_1$	-	-	-	-
DMA transfer	-	$4N(on+off)_1$	-	$N(on)_2$	-	$N(on+off)_{1,2}$	-	$N(on)_1$
Dir.up-Mem	-	-	-	-	$N(on)_2$	-	$N(on)_1$	-
MM update	-	-	-	-	$N(off)_3$	-	-	-
Total(on-chip)	**4N**	**2N**	**3N**	**N+1**	**2N**	**N**	**5N or 2N**	**N**
Total(off-chip)	**2N**	**2N**	**-**	**-**	**N**	**N**	**-**	**-**

Table 1: Number of network packets for all memory operations in cache/LM based architectures on a data set of size N, packet subindex indicates packet ordering

5.1 Based on the location of the input data

Fetch from Off-Chip: Assume a processor is about to work on a set of data as input, and assume that there is no copy of this data currently on-chip.

In the C+PP case (Figure 4.a), the processor instructs its prefetcher to bring the data set into its local cache. Each read miss in the local cache generates a request to the shared cache directory (NoC packet #1). Since we assumed that the data are not on-chip, the miss in the shared cache generates a read request to main memory (NoC pck #2; off-chip pck #1); main memory responds with the data (off-chip pck #2; NoC #3 and possibly #4). The cache hierarchy may or may not enforce the inclusion property: without inclusion(more complex), the returned data may be sent directly and only to the originally missing local cache, for a total of 3 packets through the NoC; with inclusion(less complex), the returned data must be sent to *both* caches, shared and local, for a total of 4 packets through the NoC. Note that two of the packets are short, carrying only opcode and read and return addresses, while data packets are long, carrying a cache line's data as well.



(a) Cache-based architecture

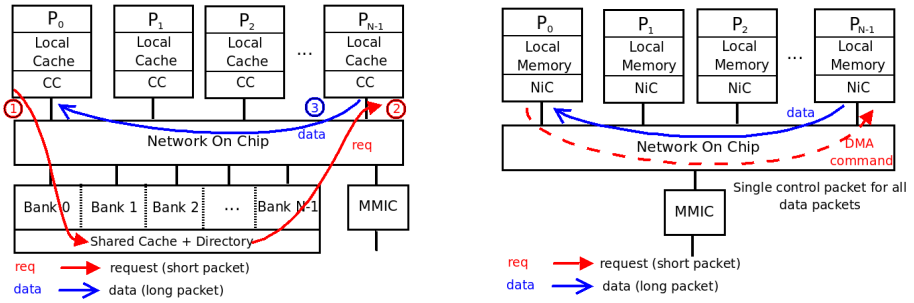
(b) LM based architecture

Fig. 4: Fetch from off-chip memory

In the LM+DMA case (Figure 4.b), the processor sets up its DMA engine to read from main memory into the local memory. The DMA engine sends one request only for N data blocks to be read (NoC and off-chip packet #1). Main memory responds with the data (NoC and off-chip pck #2). Since we assume DMA packets of size equal to a cache line each, the two systems compare as follows: off-chip traffic is the same; on-chip (NoC) traffic is 2 packets in LM+DMA versus 3 or 4 packets in C+PP, per cache line.

Fetch from another node On-Chip: Assume processor P_{N-1} in Figure 5 has produced some data which still reside in its local cache or memory, and processor P_0 decides that it will need this data and requests a local copy of them. In the C+PP system (Figure 5.a), the prefetcher of P_0 issues a read request for each cache line sought. These requests are sent to the directory, since the prefetcher does not know where the data are. The directory forwards the request to the current location of the data (P_{N-1}), and the latter sends a copy of the data directly to P_0 (and optionally updates the shared cache); the total NoC traffic is

3(or 4) packets per cache line. In the LM+DMA system (Figure 5.b), P_0 sends a remote read DMA request to P_{N-1}'s DMA engine; a *single* request packet is sent for the entire transfer. The latter engine performs a remote write DMA operation to P_0's local memory: one packet per data block travels through the NoC. For large transfers, the packet count in the two systems differs by a factor of almost 3.



(a) Cache-based architecture

(b) LM based architecture

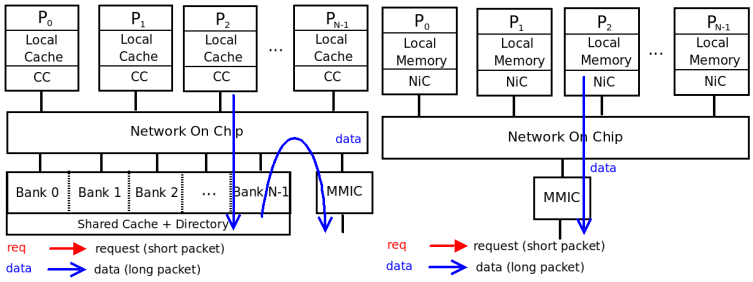
Fig. 5: Fetch from another processor's local cache/memory

Data Realignment: (gather/scatter) This operation is simply not doable with a programmable prefetcher. In the case of local memories, the DMA can be programmed to gather data from particular addresses (strided or via address list), and store them at different addresses that satisfy alignment restrictions. For example, the DMA controller can gather strided data and store them in a contiguous array, more suitable for SIMD processing. Discussing and evaluating the actual impact of this operation is left for future work.

5.2 Based on the location of the output data

Write-back or Flush to Off-Chip Assume dirty data are replaced, evicted, or flushed from a processor's local cache, and similarly, simultaneously or at a later time, from the shared cache to main memory. In Figure 6.a, in C+PP a total of 2 data packets per cache line will cross the NoC; one packet will travel off-chip. In the LM+DMA case, the processor sets up its DMA engine to copy data from local to main memory; one data packet per block will travel through the NoC and off-chip. Assuming a DMA packet carries as many data bytes as a cache line, we observe that the LM+DMA system carries half the NoC traffic, when compared to the C+PP system.

Write to another node On-Chip: Assume that processor P_0, while or just after producing some data, knows that this data will be consumed by processor P_{N-1}, and hence wants to expedite the transfer of these results.



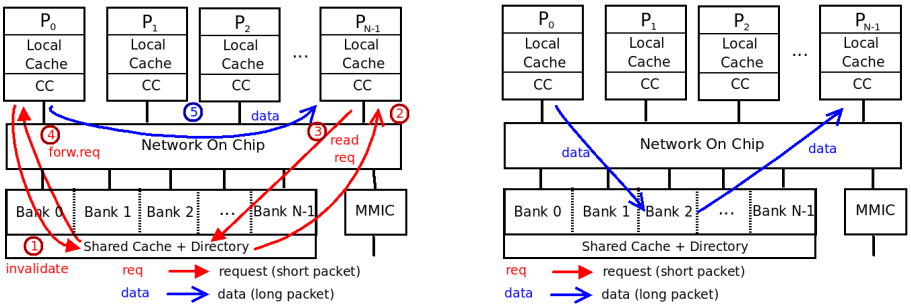
(a) Cache-based architecture (b) LM based architecture

Fig. 6: Write back or flush operation to off-chip memory

In the C+PP architecture, with the usual *write-invalidate* coherence protocol shown in Figure 7.a, there is no way for the producer to eagerly "push" its new results to the local cache of a known consumer. Even worse, if the buffer through which this communication is to occur had been used in the past for an analogous purpose, copies of the old contents may still reside in the consumer's local cache, and these old values need to be invalidated as soon as the producer writes new values into (unfortunately) its own local cache. This is illustrated in the Figure with arrows 1 and 2: these invalidation commands and their forwarding cost 2 (short) packets per cache line involved. After the producer has finished, and after proper synchronization has notified P_{N-1} to start consuming, this latter processor instructs its prefetcher to bring the new data into its local cache. This is similar to the previous case: for each cache line to be fetched, 2 short and 1 long packets cross the NoC (arrows 3, 4, 5). The grand total in this write-invalidate C+PP system is 4 short and 1 long packets per cache line communicated. To improve on this unfortunate situation, *write-update* coherence protocols have been proposed [9]. Past research has shown that write-update is usually *not* beneficial when applied to all coherence transactions, but should rather be selectively applied to only a subset of them [19]. Figure 7.b illustrates the NoC traffic under such a protocol. Assuming that all writes into a cache line are clustered in time and hence a write-combining buffer successfully collects all of them, one write-update data packet per cache line is sent to the directory, which then forwards it to the consumer's cache, for a total of 2 long packets per cache line. If writes to neighboring addresses are not clustered in time, NoC traffic will increase.

In the LM+DMA system, the local memory of the consumer is directly addressable by the producer. If neighboring writes are clustered in time, the producer may choose to send its results using remote store instructions into the consumer's local memory. If writes to various addresses occur in rather random order, the producer should rather wait until all of them have been produced, and then initiate a remote write DMA from its own to the consumer's local memory. The resulting NoC traffic will be one (long) packet per cache-line worth of data volume. There is a clear advantage by a factor of 2 to 5 in the number of

NoC packets relative to C+PP, assuming that the modified words in the DMA buffer are "dense" enough, and thus the cache-based communication would have transferred the same set of data.



(a) Write invalidate

(b) Write Update

Fig. 7: Write into another processor's local cache

6 Conclusions

Single-chip multiprocessors (CMP) have quickly become the industry trend due to their better efficiency in power, size, and performance. On-chip memories are a key factor in maintaining high performance in such CMP architectures. These on-chip memories have usually been caches; however, recent proposals used addressable local memories. In this paper, we presented a qualitative study comparing these two alternatives in terms of communication operations and network-on-chip traffic.

Cache-based architectures and prefetch mechanisms have been shown to be beneficial in cases where data communication is *implicit*. Implicit communication appears when a task does not know in advance which data it will require, or the location of such data. In these cases, hardware assistance regarding locality management and data fetch (and prefetch) works better than on-demand fetching to local addressable memories. Also, for sparse and irregular access patterns, caches have the advantage that only dirty cache lines need to be transferred. Finally, one aspect where prefetching is simpler than DMA is that prefetching does not need to be correct –nothing breaks if wrong data is prefetched– and computation does not need to explicitly synchronize with a prefetching, as is the case with DMA transfers.

On the other hand, when *explicit* communication is present, as in recent proposals in parallel programming models, the runtime system can be very efficient in exploiting local memories and on-chip DMA controllers. Software has to explicitly state which data need to be transferred, and to explicitly synchronize with the transfer. However, the runtime system can do all that in a transparent

way, since it is able to schedule task execution and data transfers ahead of time. Such scheduling can effectively hide the latency of remote data accesses, and minimizes network traffic.

The problem of using caches is the lack of scalability due to the extra traffic, and the increasing directory size imposed by the coherence protocol. In this paper, we have examined the various patterns of explicit communication, and we have found that local memories and DMA transfers reduce the amount of on-chip traffic (number of packets through the NoC) by *factors of 2 to 5*. Since the NoC only consumes energy when packets travel through it, such lighter traffic also reduces NoC power consumption by corresponding factors of 2 to 5, which is important given the significant contribution of the NoC to total chip consumption [22]. Furthermore, the communication patterns that we considered in this paper are the most optimistic ones for caches, assuming data perfectly align in the cache and there are no extra conflict misses. Careful data alignment to prevent cache conflicts constitutes a research topic by itself. The main problem of local memories is that the programmer must ensure that the task working set fits in the available on-chip space (its own local memory, or nearby memories), while caches would automatically manage moving data to and from the on-chip shared cache regardless of the dataset.

Given that both alternatives have advantages and disadvantages, we conclude that an architecture that contains both options would be suitable to a wider range of applications. The chip can have both caches and local memories, or use shared memory arrays that are configurable as either hardware caches or software addressable local memories. Integrating DMA transfers with coherent on-chip caches is an interesting issue that we will explore in future work.

References

1. P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. Memory - cellss: a programming model for the cell be architecture. In *SC*, page 86, 2006.
2. M. Byrd, G.T. Flynn. Producer-consumer communication in distributed shared memory multiprocessors. *Proceedings of the IEEE*, 87(3):456–466, Mar. 1999.
3. C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. M. Mellor-Crummey. Co-array fortran performance and potential: An npb experimental study. In *LCPC*, pages 177–193, 2003.
4. A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the performance of software distributed shared memory as a target for parallelizing compilers. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, pages 474–482, 1997.
5. A. Duran, J. Corbalan, and E. Ayguade. Evaluation of openmp task scheduling strategies. In *IWOMP*, 2008. LNCS Vol. 5004.
6. A. Duran, J. M. Perez, E. Ayguade, R. Badia, and J. Labarta. Extending the openmp tasking model to allow dependent tasks. In *IWOMP*, 2008. LNCS Vol. 5004.
7. 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. Memory - sequoia: programming the memory hierarchy. In *SC*, page 83, 2006.

8. B. Gaster. Streams: Emerging from a shared memory model. In *IWOMP*, pages 134–145, 2008. LNCS Vol. 5004.
9. D. B. Glasco, B. A. Delagi, and M. J. Flynn. Update-based cache coherence protocols for scalable shared-memory multiprocessors. In *Proc. of the 27th Hawaii Int'l Conf. on System Sciences (HICSS-27)*, volume I, pages 534–545, 1994.
10. J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *ASPLOS-III: Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 64–75, New York, NY, USA, 1989. ACM.
11. 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*, pages 297–307, New York, NY, USA, 2008. ACM.
12. J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. J. Dally. Architectural support for the stream execution model on general-purpose processors. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.
13. Intel Corporation. Write combining memory implementation guidelines. *Intel Application Notes*, 1998.
14. J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):358–368, 2007.
15. E. L. Lusk and K. A. Yelick. Languages for high-productivity computing: the darpa hpcs language project. *Parallel Processing Letters*, 17(1):89–102, 2007.
16. E. P. Markatos and M. G. H. Katevenis. Telegraphos: High-performance networking for parallel processing on workstation clusters. In *HPCA '96: Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, page 144, Washington, DC, USA, 1996. IEEE Computer Society.
17. T. Mattson. Tutorial s08 - introduction to openmp. In *SC*, page 209, 2006.
18. M. D. McCool and B. D'Amora. M08 - programming using rapidmind on the cell be. In *SC*, page 222, 2006.
19. A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. *isca*, 00:234–245, 2005.
20. S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent network interfaces for fine-grain communication. *SIGARCH Comput. Archit. News*, 24(2):247–258, 1996.
21. 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*, pages 259–272, Washington, DC, USA, 2006. IEEE Computer Society.
22. V. Soteriou, N. Eisley, H. Wang, B. Li, and L.-S. Peh. Polaris: A system-level roadmapping toolchain for on-chip interconnection networks. *IEEE Trans. VLSI Syst.*, 15(8):855–868, 2007.
23. J. Zhu, J. Hoeflinger, and D. A. Padua. A synthesis of memory mechanisms for distributed architectures. In *ICS*, pages 13–22, 2001.