NoC-Based Support of Heterogeneous Cache-Coherence Models for Accelerators

Davide Giri
Department of Computer Science
Columbia University
New York, NY, USA
davide_giri@cs.columbia.edu

Paolo Mantovani
Department of Computer Science
Columbia University
New York, NY, USA
paolo@cs.columbia.edu

Luca P. Carloni
Department of Computer Science
Columbia University
New York, NY, USA
luca@cs.columbia.edu

***Abstract*—On-chip shared memory is the primary paradigm for multi-core SoC designs and poses the most critical challenges to their scalability. Choosing the appropriate coherence model for accelerators not only can improve the overall system performance, but can also decrease energy consumption by reducing the accesses to DRAM. We propose an extension of a standard directory-based cache-coherence protocol and present its design as part of a scalable memory hierarchy implemented over a NoC. To evaluate our contribution we designed a many-accelerator SoC architecture that can support three main cache-coherence models for accelerators: non-coherent, last-level-cache-coherent, and fully-coherent. This SoC can run Linux SMP with split last-level cache and multiple DRAM controllers. Our FPGA-based experiments show that the optimal cache-coherence selection varies at run-time, based on the running accelerators and the memory footprint of the applications. Therefore, we support run-time selection of the cache-coherence model for each accelerator, as an alternative to a design-time decision.**

I. Introduction

As systems-on-chip (SoCs) integrate ever more components and become distributed systems [1], the network-on-chip (NoC) is superseding the more traditional interconnects [1]–[3]. There are many examples of this shift both in industry and academia [4], [5]. While NoCs provide more scalable on-chip communication, shared memory is the dominant programming model of multicores [6]. Shared memory and cache coherence pose critical scalability challenges, which have been widely addressed for NoC-based *homogeneous* multicores [7], [8]. Despite the NoC being identified as a proper and scalable solution also for *heterogeneous* SoCs [9], [10], the interaction between accelerators and a cache hierarchy distributed over a NoC has received limited attention by researchers.

We identified three main cache-coherence models for accelerators: non-coherent, fully-coherent, and LLC-coherent [11], [12]. In the non-coherent model the accelerator operates through direct-memory access (DMA), bypassing the caches. Conversely, with the fully-coherent model, memory requests must be coherent with the entire cache hierarchy. This approach can endow accelerators with a private cache, thus requiring no modifications to the coherence protocol. The third model represents an intermediate position: memory requests issued by the accelerator are coherent with the LLC, but not with the private caches of the processors. In this case, DMA transactions address the shared LLC, rather than external memory. Cota et al. were the first to describe the LLC-coherent model for accelerators and to estimate its benefit in simulation [11]. To date, however, neither a cache-coherence protocol nor an SoC architecture have been presented to support LLC-coherent accelerators over a NoC. To this end, we propose an extension of the MESI directory-based protocol and integrate LLC-coherent accelerators into the SoC architecture illustrated in Fig. 1, which leverages the concept of tile-based architecture over a packet-switched NoC [13]–[15]. This is the first NoC-based system that allows all three models of coherence for accelerators to coexist and operate simultaneously with support for run-time selection. Furthermore, by supporting atomic test-and-set and compare-and-swap operations over the NoC, we can run multi-processor and multi-accelerator applications on Linux SMP.

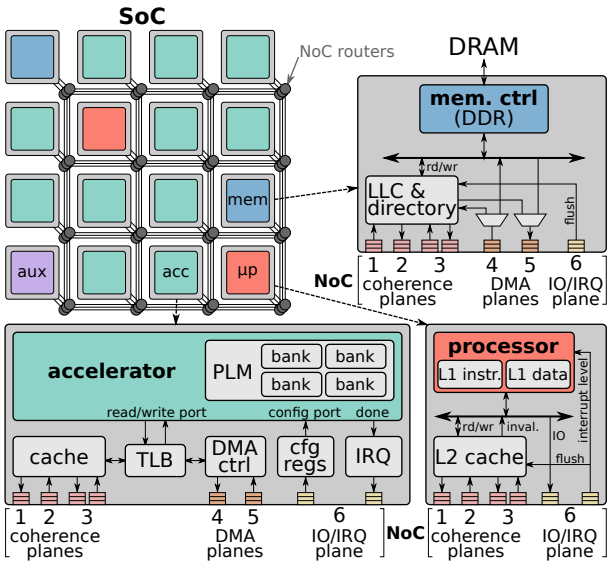

Fig. 1. 4x4 instance of our NoC-based architecture with focus on the content of accelerator, memory, and processor tiles and their interface with the NoC. Numbers 1 to 6 refer to separate physical NoC planes.

This work was supported in part by DARPA (C#: R0011-13-C-0003), the National Science Foundation (A#: 1546296) and C-FAR (C#: 2013-MA-2384), an SRC STARnet center.

With a set of experiments on FPGA, we prove the benefits of selecting the coherence model at run-time. The experimental results confirm that the LLC-coherent model significantly reduces, and in some cases eliminates, the number of accesses to main memory. In addition, LLC-coherent accelerators can have better performance than non-coherent ones, as long as the accelerated application doesn't incur thrashing of the LLC. The fully-coherent model can be the optimal selection when the memory footprint fits in the accelerator's private cache. Non-coherent DMA is optimal, instead, whenever accelerators operate on large amounts of data.

II. NoC-based Architecture

Fig. 1 shows a 4-by-4 instance of our scalable SoC. Similarly to other tile-based architectures [14], [15], each tile can host a general-purpose processor, an accelerator, or an interface with main memory. Our design is based on an instance of *Embedded Scalable Platforms (ESP)* [15], [16]. We enhanced the ESP architecture by adding a cache hierarchy to support both symmetric multiprocessing as well as loosely-coupled LLC-coherent and fully-coherent accelerators, alongside the more typical non-coherent loosely-coupled accelerators.

SoC Integration. A processor tile contains a single core and a private write-back L2 cache. The latter implements the directory-based coherence protocol over the NoC, thus decoupling the processor-specific L1-cache design from the rest of the system. For instance, our processor tile hosts a Leon3 core [17], which is tightly integrated with the write-through L1 caches. Memory requests issued over the local bus are intercepted by the L2, whereas memory-mapped I/O operations are directly forwarded to the NoC. Fig. 1 shows the local ports of the NoC planes used to route all types of messages in the system.

Memory tiles are the access points to off-chip memory. They feature a memory controller and an LLC tightly coupled with a directory. One of the memory tiles hosts shared system resources, including the interrupt controller, the system timer, and a debug interface.

The accelerator tile can host any accelerator complying with a simple interface. This consists of memory read and write ports, configuration ports, and a done signal. As shown in Fig. 1, the tile includes a set of memory-mapped configuration registers. These are accessed by the operating system through a device driver. The latter, in turn, is invoked by an application to offload a task. Some registers are accelerator specific and hold the configuration parameters, including the size of the workload. Others are common to all accelerators and hold information such as the page table handle, or the selected cache-coherence model. Based on these registers, a small TLB translates the accelerator's requests to accesses in physical memory and passes the transaction information to either the DMA controller or the private cache. We leverage a fast translation scheme based on a scatter-gather list that partitions the accelerator space in large equally-sized pages and generates a small page table [16]. Thus, accelerator tiles handle virtual memory without interrupting the processor cores.

System Invariants. For functional correctness across all coherence models, our system maintains two invariants. First, we use locks to enforce *mutual exclusion*: during the execution of any accelerator no other component can access its data. Second, during the execution of non-coherent accelerators, we ensure that there exists only a *single copy* of the data, thanks to an efficient flush mechanism. This second invariant is relaxed for LLC-coherent accelerators, because the data can be present both in DRAM and in the LLC. Indeed, most of the lines stored in the LLC are expected to be valid, if the software application was working on it before invoking the accelerator. Note that only the first invariant is necessary for fully-coherent accelerators. These can access data owned or shared by any private cache in the system, but mutual exclusion is still necessary because they do not share lock variables with the operating system and cannot perform atomic operations.

NoC Planes. We designed a packet-switched NoC with a 2D-mesh topology and look-ahead dimensional routing. In order to prevent protocol deadlock and provide sufficient bandwidth for both coherence and DMA messages, the NoC has multiple physical planes [18]. The tiles inject packets into each plane based on the type of message. Every hop takes a single clock cycle, because arbitration and next-route computation are performed concurrently. The channel width is configurable, but it is fixed at 32 bits for this work.

Directory-based protocols impose two main requirements on the interconnect to avoid deadlock: point-to point ordering and three separate channels for request, forward and response messages [19]. Hence, we devote three NoC planes to the cache coherence messages (planes 1, 2, 3 in Fig. 1). For the same reason, we route DMA requests and responses between accelerator tiles and memory tiles on two different planes (see planes 4 and 5 in Fig. 1). While we could reuse the coherence planes as DMA planes, we prefer to allocate additional ones to increase the communication bandwidth. Finally, the plane labeled *IO/IRQ* is dedicated to short packets for interrupts and memory-mapped I/O. Interrupts are not broadcasted: the interrupt controller receives all interrupts and injects in sequence one specific interrupt-level message for each processor that must be notified. While interrupt handling on the NoC inevitably incurs higher latency than on a bus, hard real-time deadlines can still be met.

Cache Design. Our cache hierarchy implements a MESI directory-based protocol over two levels of caches: a private and write-back L2 cache present on every processor tile (and, optionally, on any accelerator tile) and a combined LLC and directory that can be split across multiple memory tiles. Each partition of the LLC and directory handles requests for the same address ranges pertaining to the memory controller placed on the tile. Since the LLC and directory are tightly coupled together, we refer to them interchangeably to indicate the combination of the two. We designed the caches in SystemC and implemented them with *high-level synthesis (HLS)*. All caches are configurable in the number of sets and ways, as well as in the number of sharers and owners. For our experiments,

TABLE I
DIRECTORY CONTROLLER'S EXTENDED MESI PROTOCOL.

	REQUESTS					DMA REQUESTS		RESPONSES	
	GetS	**GetM**	**PutS**	**PutM**	**Evict**	**Read**	**Write**	**Inv-Ack**	**Data**
I	**read mem**, Excl. Data to req, owner = req / E	**read mem**, Data to req, owner = req / M	Put-Ack to req	Put-Ack to req		**read mem**, Data to req / V	**[read mem], write LLC**, / V		
V	Excl. Data to req, owner = req / E	Data to req, owner = req / M	Put-Ack to req	Put-Ack to req	**[write mem]** / I	Data to req	**write LLC**		
S	Data to req, sharers += req	Data to req, Inval. to sharers, owner = req, clear sharers / M	Put-Ack to req, sharers -= req / V (if last sharer)	Put-Ack to req, sharers -= req / V (if last sharer)	**[write mem]**, Inval. to sharers, clear sharers / I				
E	Fwd-GetS to owner, sharers+=req+owner, clear owner / S^D	Fwd-GetM to owner, owner = req / M	Put-Ack to req, if req is owner: - clear owner / V	**write LLC**, Put-Ack to req, if req is owner: - clear owner / V	Fwd-GetM to owner, clear owner / EI^D				
M	Fwd-GetS to owner, sharers+=req+owner clear owner / S^D	Fwd-GetM to owner, owner = req	Put-Ack to req	**write LLC**, Put-Ack to req, if req is owner: - clear owner / V	Fwd-GetM to owner, clear owner / MI^D				
S^D	stall	stall	Put-Ack to req, sharers -= req	Put-Ack to req, sharers -= req	stall				**write LLC**, / V (if no sharers), / S (otherwise)
EI^D	stall	stall	Put-Ack to req, sharers -= req	Put-Ack to req, sharers - = req				**[write mem]** / I	**write mem** / I
MI^D	stall	stall	Put-Ack to req, sharers -= req	Put-Ack to req, sharers -= req					**write mem** / I

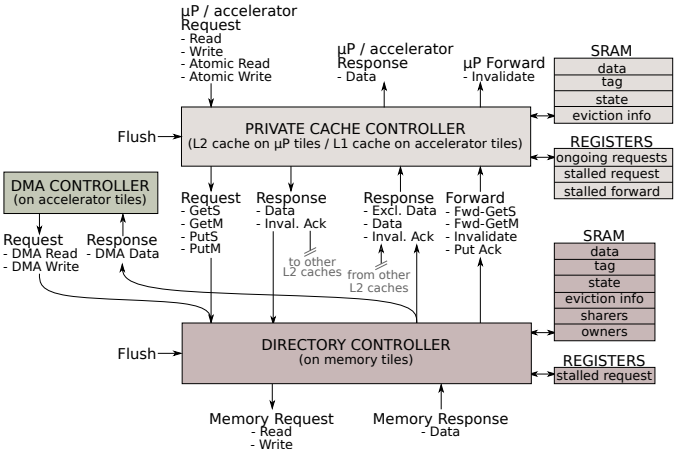

Fig. 2. Cache-coherence state and exchanged messages.

we target an FPGA but the system can be synthesized for an ASIC target as well.

Fig. 2 shows all the message types that can be sent or received by the caches. Those that involve the DMA controller are only required by LLC-coherent accelerators; they are not part of the regular MESI protocol. The cache lines and all the meta-data are stored in SRAM banks and register files. The directory controller is specified as a single SystemC process. At each iteration of the main loop, the controller checks with a fixed priority if there is any incoming message or if there is any previously stalled message that can now be processed. If this is the case, then the controller reads a whole cache set and, if needed, it updates a cache line and its meta-data. The controller might also send out one or more messages according to the protocol. The maximum latency for handling a request (i.e. for each iteration of the main loop) is exactly 4 cycles for a 16-ways LLC. The miss penalty, due to off-chip memory access, adds up to the fixed delay, when applicable. The cache storage is also implemented through banked SRAMs, offering up to 16 ports. Hence, a whole set can be read in a single clock cycle when the LLC has at most 16 ways.

The design of the private L2 cache is similar to the LLC, with the addition of a register bank that keeps all information about lines that are currently in a transient state. These registers support a configurable number of *ongoing requests*, for which the cache is only updated when a line transitions to a stable state of the MESI protocol.

We integrate the caches into their corresponding tile as follows. In the case of fully-coherent accelerators, read and write transactions are directly translated into cache-specific requests. On processor tiles, load and store operations are issued on a local bus, where the L2 cache acts as a slave device. The LLC, instead, acts as a master of a local bus on the memory tile and issues operations to address the memory controller. The flush mechanism relies on memory-mapped registers that allow the device driver to request a selective flush of any cache in the hierarchy and check for completion before invoking an accelerator. When a flush is due, the private caches wait for the processor's L1 to be flushed and then start a synchronized flush across all cache levels and all processor cores to guarantee the consistency of shared data. Note that only the levels of caches selected by the device driver are flushed and the duration of the flush phase becomes a negligible overhead when the workload of the accelerator is large enough.

III. Cache-Coherence Protocol

We modified a classic MESI directory-based cache-coherence protocol, as defined by *Sorin et al.* [19], to make it work over a NoC and, most importantly, to support LLC-coherent accelerators. The extension to the cache coherence protocol does not affect the private caches, but rather only the LLC.

A. Directory controller

Table I shows in full the new protocol for the directory controller. The format is similar to the original table [19]. The colored cells and the bold text highlight our additions and modifications. Each column corresponds to a message type that the LLC can receive. The only exception is *Evict*, which is not a message, but rather a possible consequence of a miss in the LLC. The rows, instead, represent the possible states of the cache line addressed by the incoming message. Each entry indicates the actions performed and, after the */*, the new state of the cache line. Actions within square brackets may or may not occur. The empty boxes indicate situations that never take place, while the *stall* cells require the incoming message to be stalled until the pertinent cache line resolves to a stable state. *req* refers to the requestor, which is the private cache that sent the message. *mem* stands for off-chip memory.

Write-back. First, we explicitly specify the protocol as write-back, i.e. a *Put* message does not cause a write back to main memory. Only an eviction caused by a *Get* request or a *flush* can do so. For this purpose, we add a stable state that we call *Valid* (*V*), which refers to cache lines that contain valid data but have an empty sharers' list and no owner. As shown in Table I, the only difference between *Valid* and *Invalid* is that misses for *Valid* lines do not cause memory accesses and that *Valid* lines can be evicted. We added the orange cells to describe the management of the *Valid* state. In order to explicitly define the write-back behavior, the bold fonts identify all the read and write operations to memory or to the LLC cache lines.

Dirty bit. In our implementation we only write back to memory if the cache line to be evicted is dirty, thus reducing off-chip accesses. Most of the *write mem* operations enclosed in square brackets in Table I take place for dirty lines only.

Recalls. We support sending recalls from the LLC to the private caches. These happen when a *Get* request (or a DMA request) causes a miss and there are no *Invalid* or *Valid* lines in the set. A line must be sacrificed and its sharers or owner have to be informed. *GetS* and *GetM* can only cause recalls when the LLC is not inclusive, while DMA requests may always trigger recalls. A recall requires two additional transient states corresponding to the last two rows of Table I. When a line is in either of these states, the cache is waiting for the response to a recall. The latter is sent either in the form of an *Invalidate* message, when transitioning from the *Exclusive* state, or a *Fwd-GetM* message, when transitioning from the *Modified* state. The purple cells in Table I define the behavior of recalls.

Flush. From the protocol viewpoint, a flush is a series of evictions. In our implementation, a flush only evicts *Valid* lines. This simplification, which greatly reduces the performance penalty, is possible because a flush is only needed before a non-coherent accelerator starts executing. In this situation we make sure that the flush of the private caches is completed before the flush of the LLC starts. This guarantees that all cache lines needed by the accelerator are either present in the LLC with *Valid* state, or stored in DRAM only.

DMA requests. In Table I, the green columns indicate how we extend the protocol to handle LLC-coherent DMA requests. These can only address *Valid* or *Invalid* cache lines, thanks to the invariants specified in Section II: the private caches are flushed and no other component can access the accelerator's data before completion. Note that thanks to recalls, flushing the private caches could be avoided. However, the amount of generated messages would incur a much higher traffic and performance overhead, when compared to flushing. DMA requests cause a memory access only in three possible scenarios: eviction of dirty lines, read miss, or misaligned write miss. The last condition generates at most two memory accesses, corresponding to the first and the last cache line involved in the misaligned DMA transaction. All other lines are completely overwritten and require no write allocation.

B. Private cache controller

Recalls. Recalls from the LLC are supported and implemented as forced evictions.

L1 invalidation. The processors integrated in our SoC are Leon3 cores configured with a write-through split L1 cache that supports snooping-based coherence over the AMBA AHB bus [17]. Hence, for every cache line that is evicted or invalidated in the L2, the corresponding line in the L1 is invalidated by performing a fake-write operation on the bus. Invalidation is not necessary for accelerator tiles, where the write-back L2 is the first and single level of private cache.

Atomic operations. To run an unmodified version of Linux SMP with the Leon3 processors, we support test-and-set and compare-and-swap operations. A processor issues these operations as one or more loads addressing a cache line that may or may not be followed by a store targeting the same cache line. A lock signal is set to prevent preemption of the bus. Over a NoC, standard directory-based protocols alone do not guarantee the atomicity of such operations. Hence, we add a transient state to the private cache protocol to capture the fact that a line has been read by an atomic operation, but not written yet. We call this state XM^W, which resolves to *Modified* when any of the following requests arrives on the bus: an atomic write request for the XM^W cache line, a non-atomic request, or a request for a different cache line. Additionally, when the atomic load arrives, the private cache sends a *GetM* to the directory to gain ownership of the cache line. Once it gains ownership, no forward requests are accepted for this cache line until it resolves to *Modified*. When the processor issues an atomic read request, if the related cache line is in either *Exclusive* or *Modified* state, a read hit is followed by a state update to XM^W.

TABLE II
CHARACTERIZATION OF THE TARGET ACCELERATORS.

Accelerator	Memory Footprint	PLM (kB)	FPGA Resources LUT	FF	BRAM
FFT 1D	32kB - 256kB	40	7,537	4,310	10
Sort	128kB - 4MB	24	36,868	31,300	6
FFT 2D	256kB - 16MB	128	3,965	2,190	48
SPMV	25kB - 10MB	12	8,136	4,476	24

IV. ACCELERATORS

We used SystemC and Cadence Stratus HLS to design and synthesize all loosely-coupled accelerators in our experiments. A loosely-coupled accelerator provides a major speedup over a software implementation of the same algorithm thanks to its highly parallel architecture and aggressively banked *private local memory (PLM)* [11], [20]. While the PLM occupies most of the accelerator area, typically it cannot contain the whole dataset processed by an accelerator for a given invocation. Hence, the data subsets (*chunks*) must be continuously exchanged between the PLM and the rest of the memory hierarchy. To achieve optimal performance, communication and computation phases must be overlapped as much as possible using ping-pong buffering and pipelining: the pipeline input stage loads a chunk of data into the PLM; one or more compute stages process the input data and save an output chunk into the PLM; the output stage issues a store for the partial results to the next level of the memory hierarchy. If the compute stage takes at least the same time as the input and output stages, the communication phase is completely hidden and the accelerator achieves its maximum sustainable throughput. The ability to have perfectly balanced accelerator stages is highly dependent on the specific memory access patterns, as well as on the system interconnect and the memory hierarchy, including the selected cache-coherence model. The SoC designer must consider spatial and temporal locality (if any), length of read/write transactions, and the offsets across multiple transactions, which could be statically known, or data dependent. Some accelerators, for example, never read the same data from memory twice; others, instead, may reuse data extensively. While many loosely-coupled accelerators present a streaming access pattern with long contiguous transactions, some issue irregular and short requests to memory. More importantly, the size of the dataset can vary greatly across different accelerators, and across multiple invocations of the same accelerator. Our experiments confirm that the memory footprint of the workload is indeed the most relevant metric when selecting the appropriate cache-coherence model.

We implemented four representative accelerators to carry out the experiments described in Section V. We accelerate four ubiquitous algorithms: Sort, Fast Fourier Transform (FFT) 1D, FFT 2D, and Sparse Matrix-Vector Multiplication (SPMV). The initial software implementation for SPMV is taken from the MachSuite [21]. We report below a brief qualitative description of each accelerator. Table II summarizes some quantitative data on their memory footprint and resource utilization on FPGA.

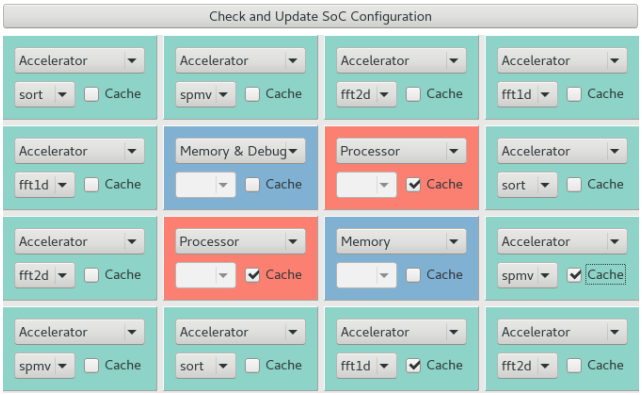

Fig. 3. SoC integrator GUI configured for the experiments.

FFT 1D accelerates the FFT algorithm over one vector of up to 32K complex numbers. The computation stages of its pipeline process two non-contiguous portions of the input vector. The offset between the two depends on the current iteration of the FFT algorithm. The number of transactions with the system memory hierarchy is proportional to the logarithm of the vector's length.

Sort can process up to 1024 vectors of 1024 floating-point numbers. Each vector fits in the PLM and it is sorted in-place. Note that no data is accessed twice and temporal locality is exploited within the accelerator's PLM.

FFT 2D operates in two phases. First FFT 1D is executed on every row of a two-dimensional matrix. While input data are read in row-major order, the output is written back in column-major order, thus transposing the resulting matrix. The second phase repeats the same operation on the transposed matrix, thus completing FFT 2D. Similarly to the FFT 1D, the number and length of the read transactions depends on the size of a row. Conversely, write transactions consist of a sequence of two-word store operations, each offset by a row.

SPMV multiplies a sparse matrix by a dense vector. The matrix is compacted in the compressed row storage format, which removes all zero entries. This dot product causes few irregular accesses to memory. The compute-to-memory ratio is very low due to the overhead of performing short read transactions compared to a simple computation stage: element-wise dot product. The elements of the matrix are read only once, while portions of the dense vector can be reused.

V. EVALUATION

For the evaluation we use an FPGA-based infrastructure built on top of the one proposed by Mantovani *et al.* [22]. Fig. 3 is a snapshot of the SoC configuration that we use for all the experiments. The CAD flow from the graphical user interface to the bitstream for FPGA is fully automated. The SoC has two Leon3 cores, two memory controllers and twelve accelerators, of which only two have a private cache (see the *Cache* selection). Through the GUI, we also select the size of the caches: $16kB$ for the L1 caches of the cores; $64kB$ for each L2 private cache; $1MB$ for each partition of the LLC, for a total of $2MB$. The bandwidth towards external

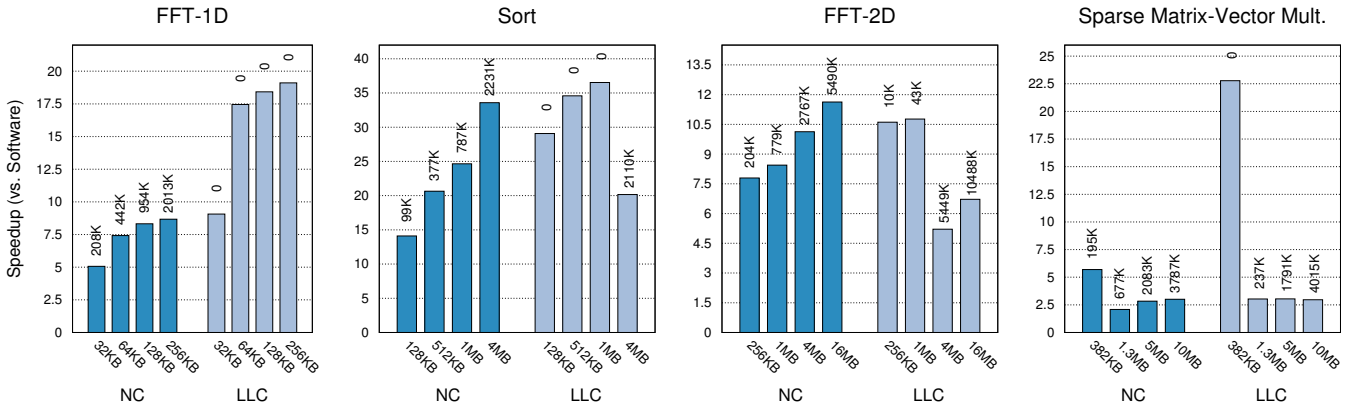

Fig. 4. Comparison of speedup w.r.t. software of non-coherent (NC) and LLC-coherent (LLC) accelerators. Bars are annotated with the memory access count.

memory is throttled by the AMBA AHB bus to one 32-bit word access per cycle. The DDR3 memory is configured to operate at its slowest possible frequency of 320 MHz. This reduction is meant to give an off-chip memory access penalty similar to that of an equivalent ASIC implementation.

A. Single-accelerator

We start by evaluating single-accelerator SoCs, to test each of the four types of accelerators in isolation. We allocate the accelerator's data on one memory partition and force the operating system to run on the other partition only. Hence, we can measure the statistics of the accelerators without the non-deterministic interference of software execution. Each test runs a user-space application that prepares the input data for the accelerator and processes its output data. Hence, caches are always hot before invoking an accelerator. Results for both non-coherent and LLC-coherent accelerators are summarized in Fig. 4. Bars represent the geometric mean of the accelerator speedup with respect to single-core software execution over several runs of the same test. For each bar, the corresponding label shows the total DRAM access count in thousands.

The charts highlight clear trends relatively to the memory footprint of the dataset. In the case of FFT-1D, Sort and FFT-2D, when the accelerator's memory footprint is smaller than the LLC size ($< 1MB$), the LLC-coherent model always has higher speedup than the non-coherent one. For larger datasets, however, the non-coherent option returns higher speedups because the LLC-coherent accelerators trigger many evictions. When considering SPMV, the charts in Fig. 4 report a much larger speedup for the smallest dataset than for the other ones. Notice that SPMV heavily benefits from LLC-coherence when the dataset fits in the LLC. The gap between the first dataset and the others for the non-coherent run is determined by the size of the dense vector only: SPMV has a 32kB PLM dedicated to the dense vector, which is used only when the vector fits in it (382KB dataset). In this case, the sparse accesses of single words to the dense vector are performed within the PLM. Because of the irregular access pattern, the LLC-coherent model continues to deliver slightly better performance, even for memory footprints larger than the LLC.

In general, despite the performance hit for large datasets, the benefits of LLC-coherence in terms of DRAM accesses are indisputable: when the dataset fits in the LLC, they are completely eliminated, with the exceptions of compulsory misses. For instance, FFT-2D operates on a temporary memory buffer, which is not accessed by software prior to invoking the accelerator. Additionally, note that the largest workloads of FFT 2D and SPMV capture the worst-case scenario of the LLC-coherent model: during these tests, most LLC-coherent DMA transactions are either a read miss that evicts a dirty line or a short misaligned write request that evicts a dirty line and doesn't write an entire cache line. Hence, each operation causes two memory accesses, as opposed to one access needed by the non-coherent DMA. In any other scenario, such as for Sort, the number of accesses to memory for LLC-coherent accelerators is always less or equal to the number of accesses required by a corresponding non-coherent accelerator.

In summary, the relative speedup of LLC-coherent accelerators, compared to non-coherent ones, ranges between 0.5x and 4x. The memory access count, instead, ranges from none to at most 2x with respect to the non-coherent model. We also observe that the non-coherent model monotonically improves performance when increasing the size of the dataset. The LLC-coherent behaves similarly, but with a jump back when the dataset becomes larger than the LLC. These results confirm that a run-time selection of the cache coherence model, based on the memory footprint of the workload, can be beneficial.

B. Many-accelerator

For a single accelerator we have shown that the effectiveness of the LLC-coherent model is strictly correlated to the ratio between the size of the workload and the capacity of the LLC. With the next set of experiments, we collect data for 4, 8 and 12 accelerators running concurrently (1, 2 and 3 instances for each accelerator type). We pick a small workload ranging from 256kB to 512kB per accelerator, such that only the aggregate dataset of 8 and 12 accelerators is larger than the LLC. For these experiments we use a dedicated memory controller for each memory partition to avoid saturation of the

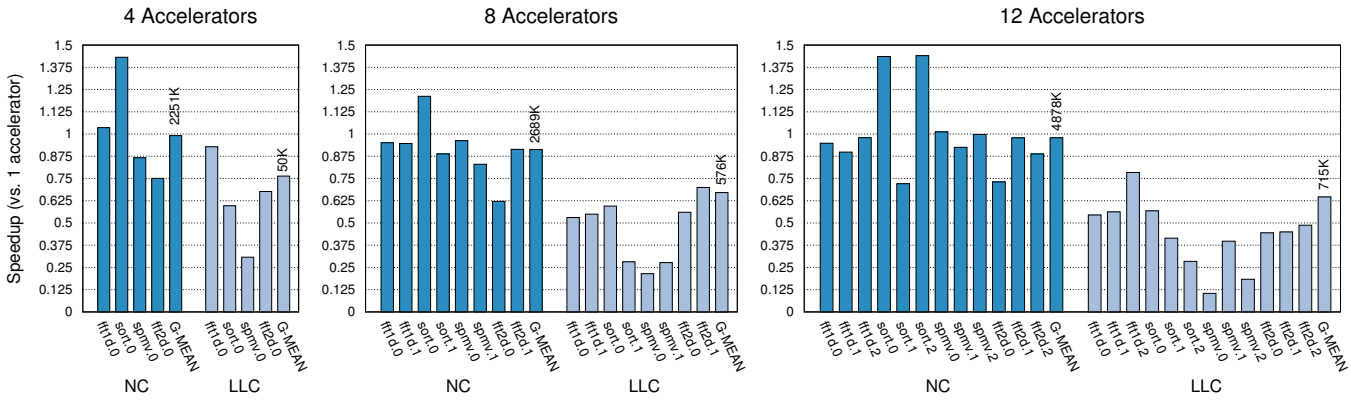

Fig. 5. Speedup of 4, 8, and 12 accelerators executing simultaneously. Each bar is normalized to the speedup of the corresponding accelerator when running in isolation. The dataset per accelerator ranges between 256kB and 512kB.

memory bandwidth. This is critical to make sure that results are not affected by under-provisioned I/O.

Fig. 5 shows the speedup of each accelerator's execution w.r.t. its execution in isolation, averaged over multiple runs. The rightmost bar on each cluster is the geometric mean of the speedups across all running accelerators. Note that the invocation of each accelerator should cause a flush of some caches depending on the coherence model. However, when a flush is issued while another one is pending, we don't re-execute it. Therefore, some accelerators benefit from a smaller invocation overhead (e.g. Sort in Fig. 5). This performance advantage would not be appreciable on large workloads, when the overhead for flushing becomes negligible.

The average performance degraded by up to 38% and 10% for LLC-coherent and non-coherent accelerators, respectively. As expected, the performance of LLC-coherent accelerators is the most penalized and the speedup degradation increases with the number of accelerators. Based on the system layout, accelerators with a dedicated path to memory perform better. In addition, accelerators operating on short and frequent transactions, like SPMV, incur larger penalties. In fact, other accelerators are likely to be granted the NoC links and lock them during long DMA transfers.

Next to performance degradation, the LLC-coherent model shows an increased number of memory accesses when running many-accelerator workloads. Nevertheless, it still maintains a considerable advantage over the non-coherent model: 44x improvement with 4 accelerators and about 5x with 8 and 12 accelerators running concurrently. As expected, these results also confirm that the selection of the cache-coherence model must account for the ratio between the workload aggregate memory footprint and the capacity of the LLC.

C. Fully-coherent model

Finally, we consider a case with a very small dataset and select the two accelerators in the system equipped with a private L2 cache. In this scenario, the fully-coherent model can have similar or better performance than the non-coherent and LLC-coherent ones. Similarly to LLC-coherent accelerators,

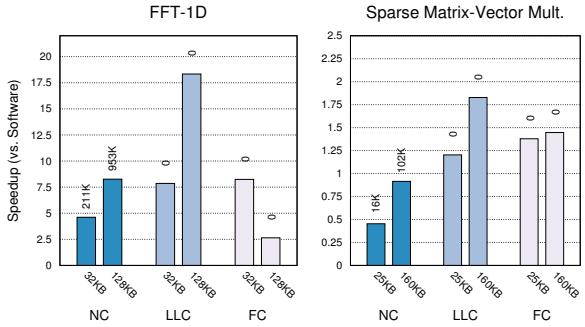

Fig. 6. Comparison of speedup w.r.t software for tiny workloads.

the fully-coherent ones have the benefit of reducing or eliminating the memory accesses. Additionally, this model does not require flushing the processors' caches, which could disrupt the work of other component of the SoC. In Fig. 6, we show side by side the accelerator speedups with respect to a software execution on a processor core for the three cache-coherence models. The fully-coherent model yields better performance only for the smallest datasets.

Given these results and since fully-coherent accelerators are widely present in the literature (e.g. [23], [24]), we support the run-time selection of the fully-coherent model as well. Our SoC generator enables this feature through the selection of an optional private cache in each accelerator tile.

VI. Related Work

Cache Coherence Models for Accelerators. What we defined as non-coherent and fully-coherent models represent the two main cache-coherence models for loosely-coupled accelerators in the literature [25]. Fully-coherent accelerators have started to receive growing interest from the industry both as off-chip [24], [26] and on-chip [27] components, but in bus-based systems only. Instead, we integrated all of the models in a NoC-based SoC.

Previous works define some bus-based variations of what we refer to as fully-coherent accelerators. These accelerators have no private cache and memory requests are issued directly on the bus. By adapting a snooping-based protocol, both the LLC

and the private caches respond to the accelerator's requests enforcing coherence [28], [29]. A similar approach over a NoC would require a costly multi-cast of invalidation and recall messages to the private caches.

Among the proposed solutions to support non-coherent accelerators over a NoC, some suggest to keep a separate memory space for the accelerators [9], while more recent approaches agree on maintaining shared memory to avoid copying data across the two address spaces [10], [30]. Our implementation follows the most recent approach.

The few studies that compare cache-coherence models for accelerators differ with respect to our work in that they study bus-based systems, experiment on single-accelerator workloads and do not compare all three models that our architecture supports. With *Fusion*, Kumar et al. presented three designs of the fully-coherent model [23]. Shao et al. analyzed the non-coherent and fully-coherent models [12]. Finally, Cota et al. evaluated LLC-coherent and non-coherent accelerators [11]. While these works rely mostly on simulation, our study is based on FPGA implementations. This allows us to run complex multi-threaded applications, on top of Linux SMP, that invoke multiple accelerators operating on large workloads.

Similarly to *Fusion*, other works explored the case of multiple accelerators that share the same private L1 cache or scratchpad [9], [10]. Arguably, a group of accelerators sharing the same L1 cache or PLM can be defined by its aggregate communication pattern and workload size. Hence, our conclusions would still apply.

Cache Hierarchy and NoC Optimization. Researchers proposed several ways to optimize the cache hierarchy in NoC-based multicores. However, the implication of cache-coherence over a NoC for accelerators has received limited attention. Some approaches for homogeneous multicores propose a modification of the directory-based protocol [8], [31]. Others suggest to restructure the interconnect and embed the cache coherence protocol in the NoC, thus completely removing the directory [7]. Alternatively, Cong et al. [32] propose a hybrid NoC interconnect as the backbone for many-accelerator architectures. These types of optimization are orthogonal to our work. Our protocol can be implemented on different types of NoC. In fact, the only restrictions that apply to the network are point-to-point ordering and the availability of three distinct planes, or virtual channels.

VII. Conclusion

We proposed an extension to the MESI directory-based cache coherence protocol over NoC to support LLC-coherent accelerators. We presented the first NoC-based SoC enabling non-coherent, LLC-coherent and fully-coherent accelerator models to coexist and operate simultaneously. By implementing atomic test-and-set and compare-and-swap, our SoC can run complex accelerated applications on top of Linux SMP. Experiments on FPGA prove the importance of run-time selection of the accelerator coherence model. In particular, the results show how supporting LLC-coherent accelerators can deliver up to 4× the performance of non-coherent accelerators, while reducing (or in some cases eliminating) the number of accesses to external memory.

References

[1] L. P. Carloni, "From latency-insensitive design to communication-based system-level design," *Proc. of the IEEE*, 2015.

[2] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," *IEEE Computer*, 2002.

[3] P. Pande *et al.*, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures," *IEEE Trans. on Computers*, 2005.

[4] L. Case, "Easing heterogeneous cache coherent SoC design using Arteris Ncore interconnect IP," *The Linley Group*, 2016.

[5] J. Balkind, "OpenPiton: an open source manycore research framework," in *Proc. of ASPLOS*, 2016.

[6] M. Martin *et al.*, "Why on-chip cache coherence is here to stay," *Communication of ACM*, 2012.

[7] W. Kwon and L. Peh, "A universal ordered NoC design platform for shared-memory MPSoC," in *Proc. of ICCAD*, 2015.

[8] E. Bolotin *et al.*, "The power of priority: NoC based distributed cache coherency," in *Proc. of NOCS*, 2007.

[9] M. Lyons *et al.*, "The accelerator store: A shared memory framework for accelerator-based systems," *Trans. on Architecture and Code Optimization*, 2012.

[10] Y. T. Chen *et al.*, "Accelerator-rich CMPs: from concept to real hardware," in *Proc. of ICCD*, 2013.

[11] E. Cota *et al.*, "An analysis of accelerator coupling in heterogeneous architectures," in *Proc. of DAC*, 2015.

[12] Y. Shao *et al.*, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," in *Proc. of MICRO*, 2016.

[13] W. J. Dally and B. Towles, "Route packets, not wires: On-chip inteconnection networks," in *Proc. of DAC*, 2001.

[14] J. Cong *et al.*, "Accelerator-rich architectures: Opportunities and progresses," in *Proc. of DAC*, 2014.

[15] L. P. Carloni, "The case for Embedded Scalable Platforms," in *Proc. of DAC*, 2016.

[16] P. Mantovani *et al.*, "Handling large data sets for high-performance embedded applications in heterogeneous systems-on-chip," in *Proc. of CASES*, 2016.

[17] J. Gaisler, "An open-source VHDL IP library with plug & play configuration," *Building the Information Society*, 2004.

[18] Y. J. Yoon *et al.*, "Virtual channels and multiple physical networks: Two alternatives to improve NoC performance," *IEEE Trans. on CAD*, 2013.

[19] D. Sorin *et al.*, *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool, 2011.

[20] C. Pilato *et al.*, "System-level optimization of accelerator local memory for heterogeneous systems-on-chip," *IEEE Trans. on CAD*, 2017.

[21] B. Reagen *et al.*, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proc. of IISWC*, 2014.

[22] P. Mantovani, *et al.*, "An FPGA-based infrastructure for fine-grained DVFS analysis in high-performance embedded systems," in *Proc. of DAC*, 2016.

[23] S. Kumar *et al.*, "Fusion: design tradeoffs in coherent cache hierarchies for accelerators," in *Proc. of ISCA*, 2015.

[24] S. Neuendorffer and F. Martinez-Vallina, "Building Zynq® accelerators with Vivado® high-level synthesis," in *Proc. of Symp. on FPGA*, 2013.

[25] Y. Shao and D. Brooks, *Research Infrastructures for Hardware Accelerators*. Morgan & Claypool, 2015.

[26] J. Stuecheli, "POWER8," in *Proc. of the IEEE Hot Chips Symp.*, 2013.

[27] H. Franke *et al.*, "Introduction to the Wire-Speed processor and architecture," *IBM J. Research & Development*, 2010.

[28] J. Goodacre, *The Effect and Technique of System Coherence in ARM Multicore Technology*. MPSoC, 2008.

[29] B. Blaner *et al.*, "IBM POWER7+ processor on-chip accelerators for cryptography and active memory expansion," *IBM J. Research & Development*, 2013.

[30] Y. Hao *et al.*, "Supporting address translation for accelerator-centric architectures," in *Proc. of HPCA*, 2017.

[31] Y. Yao *et al.*, "SelectDirectory: A selective directory for cache coherence in many-core architectures," in *Proc. of DATE*, 2015.

[32] J. Cong *et al.*, "On-chip interconnection network for accelerator-rich architectures," in *Proc. of DAC*, 2015.