Lightweight hardware support for selective coherence in heterogeneous manycore accelerators

Alessandro Cilardo, Mirko Gagliardi, and Vincenzo Scotti
University of Naples Federico II and CeRICT, Naples, Italy, contact email: acilardo@unina.it

***Abstract*—Shared memory coherence is a key feature in manycore accelerators, ensuring programmability and application portability. Most established solutions for coherence in homogeneous systems cannot be simply reused because of the special requirements of accelerator architectures. This paper introduces a low-overhead hardware coherence system for heterogeneous accelerators, with customizable granularity and noncoherent region support. The coherence system has been demonstrated in operation in a full manycore accelerator, exhibiting significant improvements in terms of network load, execution time, and power consumption.**

I. Introduction

Accelerator-based –or heterogeneous– computing has become increasingly important in a variety of scenarios [1], ranging from High-Performance Computing (HPC) to embedded systems. Heterogeneous resources may include GPUs but also dedicated processing units, often implemented on FPGAs or dedicated ASIC units. To maximize resource and power efficiency, accelerator architectures tend to rely on parallelism to improve performance, with multi/manycore accelerators being today commonplace. In such scenarios, coherent shared memory is an important facility [2] acting as a key enabler for programmer-friendly models exposed to the software as well as for the effective adaptation of existing parallel applications. However, unlike general-purpose architectures, hardware-managed coherence poses a major challenge for accelerators, due to the cost of the coherence infrastructure as well as the possible limitations in terms of scalability and performance. Full implementation of standard coherence protocols can induce significant overheads even when there is essentially no data sharing, e.g. when handling a nonshared block eviction. In fact, in many workloads a significant fraction of blocks are private to a single processing unit requiring in principle no coherence maintenance [3], [4]. While such problems have been widely investigated in the area of conventional homogeneous architectures, manycore accelerator-based systems pose special requirements and constraints. For example, some existing solutions require a software support [5] or even an active role of the Operating System in conjunction with a paging mechanism [4], which cannot be assumed on most accelerator-based systems. Other approaches target small/medium-scale on-chip multi-processors (CMPs) and rely on broadcasting mechanisms [6] [7], limiting the scalability towards massively parallel manycore systems.

This paper introduces a low-overhead hardware coherence system for heterogeneous accelerators, with customizable granularity and noncoherent region support, able to significantly reduce the messaging overhead due to coherence transactions. The coherence maintenance system distinguishes private and shared data, avoiding unnecessary coherence operations and optimizing indirection latencies for private data. The architecture is mostly protocol-independent and is demonstrated here in conjunction with an extended Modified-Shared-Invalid (MSI) protocol.

Some related contributions in the technical literature are briefly reviewed in Section II, while Section III provides the details of the proposed coherence infrastructure and Section IV presents the results of our experimental evaluation. Unlike typical works dealing with coherence in homogeneous processors, these results are not only collected from simulation, as the technique has been embodied in a fully-fledged manycore accelerator available as an RTL model and emulated on a large-scale FPGA platform. The evaluation in Section IV, which relies on a number of kernels with various compute/memory access patterns, shows significant reduction of the communication load and, consequently, improved execution time and energy consumption. The results also highlight the limited hardware overhead incurred by the proposed technique and confirm its potential benefits for future large-scale manycore accelerators, as summarized by the final remarks in Section V.

II. Related work

Most of the relevant work in the literature targets small/medium scale CMP architectures relying on snooping protocols. For example, in [6] the memory space is divided in regions and each CMP core keeps track of sharing information with a coarse granularity. Broadcasting on the shared bus is used to keep this information consistent across the system, while coherence transactions can be significantly reduced in case of exclusive ownership of memory blocks. The technique does not address false sharing, with cores accessing different parts of the same block still competing for its ownership, and extensively relies on broadcasting, which limits the scalability of the solution. Similar considerations hold for [8], where a table called Region-Coherence Array is deployed to track the memory region's coherence state.

Demetriades et al. [9] propose Stash Directory a solution which extends the traditional sparse directory by avoiding invalidation for blocks that are known to be private. Our solution takes a similar approach, although in our work the directory is totally unaware of private blocks, so both directory entry evictions and directory indirection are avoided.

The work in [7] eliminates both the traditional coherence invalidation/update scheme and the costly sharer-tracking mechanism, mitigating the directory storage overhead, the need for indirection as well as the traditional protocol complexity. The solution, however, still relies on an expensive broadcast-based invalidation mechanism to retrieve all block copies from the sharers.

Power et al. [1] focus on hardware coherence in CPU-GPU-centric heterogeneous systems, highlighting coherence bottlenecks and showing that limited directory resources are

978-3-9819263-2-3/DATE19/©2019 EDAA

a major show-stopper in such systems. In fact, a few solutions aim to reduce the directory size by redefining the granularity of the coherence regions. SCT [10] supports a dual-grain coherence system which tracks private regions by observing memory requests from the cores and keeping only one entry at the directory level for any number of blocks in that region.

In [5], Kelm et al. propose a hybrid solution for switching from a hardware coherence maintenance scheme to a software model, and vice versa. The solution heavily relies on a global coherence table within the last-level cache (LLC), which tracks the coherence approach to use and the block-level granularity. Cuesta et al. [4] aim to improve the efficient use of the memory in the directory. The authors propose a mechanism that classifies memory blocks into private and shared data, providing coherence support only for shared blocks. The technique relies on memory paging support and an active involvement of the Operating System, and hence it does not lend itself for the case of bare-metal accelerators.

III. Proposed coherence infrastructure

Before presenting our coherence system, we provide a few technical details of the manycore accelerator used for our experiments. We relied on the Naples Processing Unit (Naples PU) [11] open-source manycore system, demonstrated by the authors within the MANGO H2020 projcet. Naples PU is based on a 2D mesh of heterogeneous tiles relying on a network-on-chip (NoC). The NoC routers are tightly coupled with network interface modules providing packet-based communication over four different virtual channels. A two-stage look-ahead router is used implementing a wormhole flit-based communication. The networking infrastructure allows both intra-tile and inter-tile communication. One virtual channel is dedicated to service message flows. In particular, the manycore system supports a distributed synchronization mechanism based on hardware barriers. The accelerator core, shown in Figure 1a, is fully parameterizable. It features a lightweight control infrastructure, hardware multithreading as well as a vector instruction set targeted at data-parallel kernels, a typical approach taken by heterogeneous accelerators. In particular, the core is organized in N processing elements (PEs), each capable of both integer and floating-point operations on independent data. Correspondingly, each thread is equipped with a vector register file, where each register can store up to N scalar data allowing each thread to perform vector operations on N independent data simultaneously. In the default configuration, the architecture employs $N = 16$ PEs that can handle 32-bit data concurrently. The accelerator core also features its own m-way set-associative write-back L1 caching system, as well as a load/store unit tightly interconnected with the cache controller. The load/store unit is independent of the adopted coherence protocol. For each cached block it stores two permission bits, namely `can_read` and `can_write`, which are updated by the cache controller and used by the accelerator to detect a cache hit or miss. The cache line width matches the number of physical PEs, i.e. a read memory request loads 16 scalar data from the main memory and stores them into a vector register at once. Each core stores a byte-level dirty mask for each private cache block, which is evaluated when the cache line is flushed back to the LLC. This mask is attached to the message and the LLC proceeds to update only the dirty part of the cached value. Such a mechanism allows multiple cores to work on non-overlapping portions of the same memory blocks without incurring any data loss and unneeded contentions, providing an effective solution to the false sharing problem.

A. Baseline coherence system

To ensure scalability, Naples PU resorts to a sparse directory approach and a distributed L2 cache, resulting in a dramatic area reduction compared to a full-map directory model [12]. Each tile deploys a coherence maintenance infrastructure along with the accelerator core, as shown in Figure 1b. A cache controller handles the local processing unit's memory requests, turning load and store misses into directory requests over the network. It also handles responses and forwarded requests coming from the network, updating the block state in compliance with the given coherence protocol, while the accelerator is totally unaware of it. In fact, the architectures of the cache and the directory controllers have been designed with flexibility in mind and are not bound to any specific coherence protocol. They are equipped with a configurable *protocol ROM*, which provides a simple means for coherence protocol extensions, as it precisely describes the actions to take for each request based on the current block state. Furthermore, the directories are totally unaware of the selective coherence deactivation for private blocks, described in Section III-B, so any existing design can be used.

B. Selective coherence deactivation

Our solution supports selective coherence deactivation for private data by means of noncoherent memory region tables, which track the start/end addresses of noncoherent areas. The granularity of a region is configurable. It is set to 4 MB in the default configuration. When an accelerator requests a memory access, if the requested address lies within any of the configured noncoherent regions, the local cache controller is notified that this is a noncoherent memory access. The location of the table impacts the overall system performance. While a single global table (which can be distributed uniformly in the design, as in [5]) ensures a consistent view of the memory configuration, the corresponding operations (updates, queries, etc.) cause additional pressure on the interconnection. In our solution, each tile is equipped with a private table, minimizing the table access latency, under the assumption that the initial table configuration by the software is consistent. Consequently, operations on the table generate no additional traffic over the network-on-chip. The region tables can be accessed by the accelerator through special control registers, as shown in Figure 1c, providing a flexible configuration and debugging infrastructure. Registers are directly mapped to the region table interface, allowing entry allocation/modification from the accelerator side during run-time.

C. Extended MSI protocol

The basic MSI coherence protocol has been extended to support non-coherent memory blocks. The cache controller is the only coherence actor aware of the noncoherent memory blocks, as the directory controller is completely bypassed in case of noncoherent accesses and thus it allocates no entry during noncoherent transactions. The proposed protocol is depicted in Figure 1d in a simplified diagram which involves the new noncoherent states and the related transient states. Every request coming from the local accelerator is tagged with a coherence bit, according to the region table look-up result.

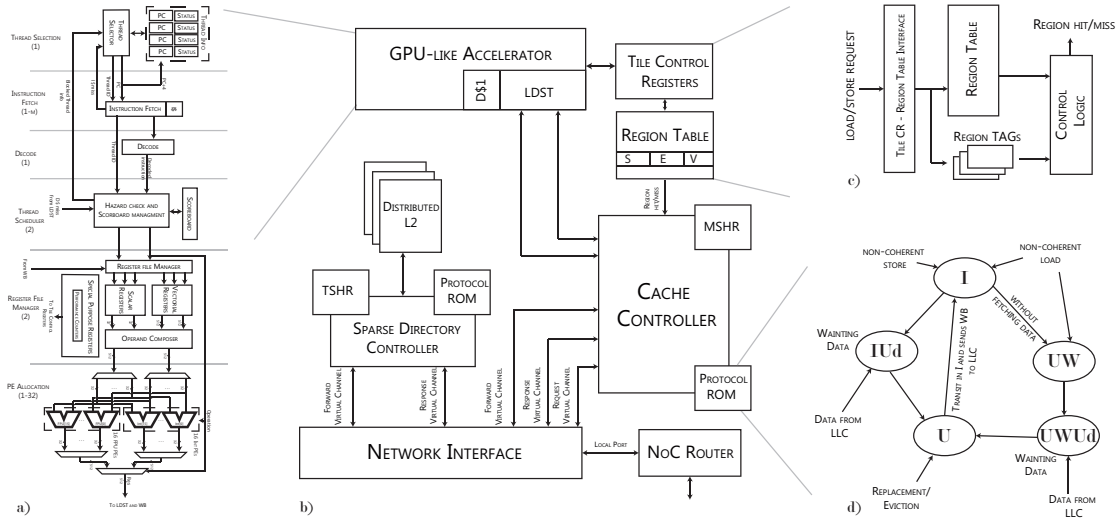

Fig. 1. a) Overview of an accelerator core within Naples PU. b) Detail of a processing tile extended with the proposed coherence system. This figure highlights the Cache Controller (CC) and the extension for noncoherent region support. The CC mainly relies on the distributed directory, on an extended coherence protocol which resides in its embedded protocol ROM, and on a local bypass which allows the CC to directly access the forward virtual network interface of the network infrastructure. c) Detail of the region look-up table. d) Extended MSI protocol used in the Cache Controller. Only the noncoherent states and the related transient states are represented.

When a block is in the invalid state `I`, the first access will determine which coherence mechanism will be applied on it. If the access is a noncoherent load, the read request is forwarded directly to the LLC while the block is in the transient state `IUd` waiting for the data. The noncoherent state, namely `U`, is applied to this block when data are received. All the subsequent reads or writes will always result in a cache hit and no additional traffic is required to track the block status. On the other hand, if the first access is a noncoherent store, the request always results in a hit, the block transits into the noncoherent write state, namely `UW`, and no memory block is fetched form the LLC. A bit-mask is used to keep track of which bytes of the block are dirty: further loads from those parts will result in a hit. This approach is based on the observation that store-first noncoherent blocks are usually used to track either the output of the computations or the memory stack of a core (which is also private), so there is no need to fetch their previous value. This significantly reduces the overall network traffic, since no message is generated during these operations. If a noncoherent load request occurs for a block in the `UW` state, the cache controller checks the dirty bit-mask. If the request address offset is marked as dirty, the data is retrieved from the local cache and no coherence state transition occurs. On the other hand, if the requested data is not marked as dirty, it needs to be fetched from the LLC, so the cache controller sends the request over the network-on-chip and moves the block state from `UW` to the transient state `UWUd`. As soon as the data is received, the block transits into state `U`. Notice that our solution does not rely on broadcasting for eviction, thereby improving performance and network efficiency.

IV. Experimental evaluation

The proposed approach has been fully validated and demonstrated on the Naples PU system, as an example of a real large-scale manycore accelerator, emulated on the FPGA-based platform provided by the MANGO project. Naples PU also comes with an LLVM-based software toolchain, not described here, used to compile the parallel kernels under study, which are written in C language. The experimental system features a 4×4 mesh with 14 units, one memory controller tile, and one host-communication tile. Each unit deploys 16 PEs and it is equipped with 8 hardware threads, totalling 224 PEs physically allocated in the system. Threads in the same accelerator share the same L1 cache and network access interface. The quantitative evaluation was carried out on a proFPGA MB-4M FPGA board by ProDesign, equipped with one Xilinx Virtex-7 XC7V2000T FPGA. Each kernel used for the evaluation is parallelized over 128 processing elements. The host-communication tile is used to gather termination signals from the parallel threads and determine the completion of the kernel. Each workload is executed (1) using a baseline MSI coherence support, and (2) employing our selective coherence system. In the latter configuration, the noncoherent regions can overlap with the kernel input and output data. We first measure the impact of our solution on the network traffic in terms of the total number of flits processed by the routers. Figure 2c shows the results. By overlapping the noncoherent regions with the kernel data section, we observe a remarkable saving in terms of flits flowing over the network (up to 77% in the `convolution` workload). This is due to the lower number of unnecessary coherence requests sent to the involved directory controllers, resulting in both a considerable reduction of the indirection latencies and limited transaction overhead caused by false sharing. Next, we evaluate the impact of the proposed mechanism in terms of total kernel clock cycles and the number of data misses occurred during the computation. These results are shown in Figure 2b and Figure 2a. The number of data cache misses drops in all the presented workloads, almost up to 80% in the `dct`, and about 85% in the `kmeans` workload, while in `convolution`, `matrix`

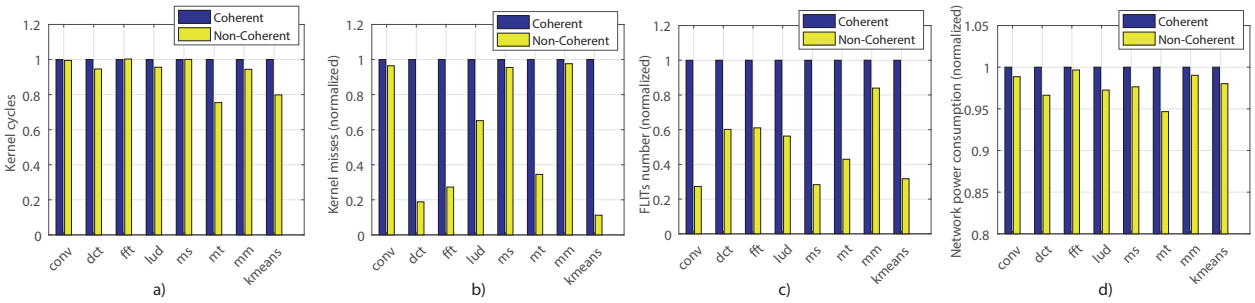

Fig. 2. a) Number of cycles for each kernel in coherent and noncoherent configurations. b) Total number of data misses in the whole system. c) Total number of flits flowing through the network-on-chip. d) Dynamic power consumption of the networking infrastructure.

`multiplication`, and `marching squares` these numbers stay constant. Furthermore, in `dct`, `fft`, and `kmeans` these reductions are more prominent due to the distribution of the data. Such kernels are heavily affected by false sharing due the granularity of the memory blocks. In these cases, different PEs compete for different data placed within the same memory blocks, resulting in coherence maintenance which generates unnecessary network messages when running with baseline MSI coherence support. In terms of kernel duration, we observe a reduction in most of the presented workloads, up to 25% for `matrix transpose`. In three cases our solution has a marginal impact, namely `convolution`, `fft`, and `marching squares`. Although these workloads generate less flits, the memory layout for noncoherent data and the accelerators' multithreading support hide the potential improvement. Finally, we evaluate the dynamic power reduction in the networking infrastructure. Figure 2d shows the results. All workloads experience a power reduction, up to 5% in the `matrix transpose` case. These results are directly correlated to the previously exposed results, i.e. the reduction of flits and directory activity.

TABLE I. RESOURCE OCCUPATION OF ONE TILE.

	LUT	Flip-Flop	BRAM
Baseline coherence system	20.888 (*135.717*)	43.197 (*216.508*)	37 (*133*)
Selective coherence system	24.045 (*132.560*)	46.308 (*213.397*)	37 (*133*)

A. Hardware overhead

We also discuss the hardware overhead incurred by the coherence system, both in the baseline version and the version proposed here supporting multi-grain blocks, selective coherence deactivation, and private data optimizations. In both cases the system can handle up to 32 concurrent coherence transactions, while the Region Tables contain 128 entries. Table I shows the hardware resources required by the emulated system on the Virtex-7 XC7V2000T FPGA. The numbers refer to the additional hardware needed for coherence maintenance in a single manycore tile, while the values in parentheses indicate the overall requirements of a tile. The baseline coherence support brings a moderate overhead on the tile cost, and proportionally on the whole manycore, ranging from 16% to 28% depending on the type of resources, while most of the hardware is used for processing. The selective coherence support proposed here incurs a marginal additional cost compared to the baseline version, as low as 13% for LUTs and 6% for FFs.

V. CONCLUSIONS

Hardware-managed coherence is an important feature for emerging large-scale manycore accelerators. The design proposed here provides a lightweight, scalable solution for selective coherence, enabling significant improvements in terms of network load, execution time, and power consumption.

Acknowledgments. This work is supported by the European Commission in the framework of the H2020-FETHPC-2014 project n. 671668 - MANGO: exploring Manycore Architectures for Next-GeneratiOn HPC systems.

REFERENCES

[1] J. Power *et al.*, "Heterogeneous system coherence for integrated CPU-GPU systems," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 457–467.

[2] M. M. Martin *et al.*, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[3] N. Hardavellas *et al.*, "Reactive NUCA: near-optimal block placement and replication in distributed caches," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 184–195, 2009.

[4] B. Cuesta *et al.*, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *38th Annual Int. Symposium on Computer Architecture (ISCA)*, June 2011, pp. 93–103.

[5] J. H. Kelm *et al.*, "Cohesion: a hybrid memory model for accelerators," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 429–440.

[6] A. Moshovos, "Regionscout: Exploiting coarse grain sharing in snoop-based coherence," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2, pp. 234–245, 2005.

[7] S. H. Pugsley *et al.*, "SWEL: Hardware cache coherence protocols to map shared data onto shared caches," in *Procs of the 19th Int. Conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 465–476.

[8] J. F. Cantin *et al.*, "Improving multiprocessor performance with coarse-grain coherence tracking," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2. IEEE Computer Society, 2005, pp. 246–257.

[9] S. Demetriades and S. Cho, "Stash directory: A scalable directory for many-core coherence," in *High performance computer architecture (hpca), 2014 ieee 20th Int. Symposium on*. IEEE, 2014, pp. 177–188.

[10] M. Alisafaee, "Spatiotemporal coherence tracking," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 341–350.

[11] "The Naples Processing Unit (Naples PU)," http://www.NaplesPU.org, accessed: 2018-11-29.

[12] A. Gupta *et al.*, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Scalable shared memory multiprocessors*. Springer, 1992, pp. 167–192.