Runtime Reconfigurable Memory Hierarchy in Embedded Scalable Platforms

Davide Giri
Columbia University
New York, USA
davide_giri@cs.columbia.edu

Paolo Mantovani
Columbia University
New York, USA
paolo@cs.columbia.edu

Luca P. Carloni
Columbia University
New York, USA
luca@cs.columbia.edu

ABSTRACT

In heterogeneous systems-on-chip, the optimal choice of the cache-coherence model for a loosely-coupled accelerator may vary at each invocation, depending on workload and system status. We propose a runtime adaptive algorithm to manage the coherence of accelerators. The algorithm's choices are based on the combination of static and dynamic features of the active accelerators and their workloads. We evaluate the algorithm by leveraging our FPGA-based platform for rapid SoC prototyping. Experimental results, obtained through the deployment of a multi-core and multi-accelerator system that runs Linux SMP, show the benefits of our approach in terms of execution time and memory accesses.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Hardware → Hardware accelerators**; **Hardware-software codesign**;

KEYWORDS

heterogeneous system-on-chip, hardware accelerators, cache coherence, FPGA prototyping

ACM Reference Format:
Davide Giri, Paolo Mantovani, and Luca P. Carloni. 2019. Runtime Reconfigurable Memory Hierarchy in Embedded Scalable Platforms. In *Proceedings of ASPDAC '19: 24th Asia and South Pacific Design Automation Conference (ASPDAC '19),* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3287624.3288755

ASPDAC '19, January 21–24, 2019, Tokyo, Japan

ACM ISBN 978-1-4503-6007-4/19/01…$15.00
https://doi.org/10.1145/3287624.3288755

1 INTRODUCTION

As the need for hardware acceleration has become imperative, heterogeneous systems-on-chip (SoCs) have quickly earned a fundamental role across all computing platforms [2, 12, 31]. Heterogeneity brings energy efficiency and high performance, but it complicates the task of system integration of components in terms of hardware-software co-design, access to shared memory and design regularity [8]. In fact, the integration of loosely-coupled accelerators in an SoC presents hard design challenges, especially regarding the interaction with the memory hierarchy. The literature shows at least three different cache-coherence models for accelerators, which we refer to as *fully-coherent*, *LLC-coherent* and *non-coherent* [11, 15, 26]. In the non-coherent model, accelerators access off-chip memory directly, bypassing the cache hierarchy. Fully-coherent accelerators are coherent with the private caches of the processor cores. LLC-coherent is an intermediate approach, where the accelerators are only coherent with the last-level cache (LLC), but not with the private caches in the system.

In a prior analysis we showed that the optimal choice of coherence model for an accelerator varies at runtime [15]. Preliminary results suggest that the best model depends on both static and dynamic factors: the memory access pattern of the interested accelerator, the status of the system containing it, and the software application invoking it. For this reason, we developed a scalable architecture that relies on a network-on-chip (NoC) to provide support for runtime selection of the cache-coherence model and for the coexistence of heterogeneous models in an SoC [16]. In this paper we show how to leverage such heterogeneity to improve the overall system performance. We propose a lightweight runtime algorithm to adaptively select the cache-coherence model for each accelerator's invocation in heterogeneous SoCs. In addition to information on SoC floorplanning and static characteristics of the accelerator's memory access pattern, the algorithm makes decisions based on the current status of the system and on the size of the accelerator's memory footprint, which can change at each invocation.

To evaluate the algorithm, we leveraged our rapid FPGA prototyping platform for SoC design, part of the Embedded Scalable Platforms (ESP) project [8, 21]. Our evaluation SoC is composed of two CPUs, two memory controllers and twelve

loosely-coupled accelerators, all connected by a multi-plane NoC. The system boots Linux SMP and can run the synthetic application on top of it.

We designed a synthetic accelerator that can be configured to behave with a wide range of memory access patterns. We then placed twelve differently configured instances of this accelerator in our evaluation SoC. Additionally, we built a synthetic multi-threaded application that invokes the twelve accelerators. The application is divided in nine phases, where each phase presents a different combination of workload sizes and maximum number of concurrently active accelerators.

We first executed the synthetic application with a fixed and homogeneous choice of coherence model, one execution for each of the three coherence models to obtain three baseline configurations. Then we run the application with our proposed algorithm. The experimental results show that, in each phase of the application, the runtime algorithm always performs at least as well as the best performing among the three baselines, with up to a 35% improvement. Accordingly, the algorithm leads to the best execution time over the whole application. With respect to the number of memory accesses, our algorithm is always better than the non-coherent baseline, while it has more accesses than LLC-coherent and fully-coherent in some phases as a trade-off to reduce the execution time. Our approach has no hardware cost and the software overhead is limited to the execution of the algorithm at each accelerator's invocation. The algorithm accounts for only 6% of the lines of code of device driver.

2 CACHE-COHERENCE MODELS

Loosely-coupled accelerators are known for providing major speedups and energy savings, thanks to a highly parallel custom datapath and an aggressively banked *private local memory* (*PLM*) [11]. Although highly configurable, these accelerators are not programmable, i.e. they do not execute instructions. They do not require a fine-grained synchronization with the processor cores. A device driver invokes the accelerator and is notified back via interrupt when the accelerator has completed its task. In fact, loosely coupled accelerators are normally invoked to execute coarse-grained tasks. They also require exclusive access to their data structures during execution, a common programming requirement easily enforced with Linux.

Let's consider a multi-accelerator SoC, like for instance the one shown in Figure 1. Which cache-coherence model should be used for the accelerators? The different solutions proposed in literature can be grouped into three main classes: *fully-coherent*, *LLC-coherent* and *non-coherent* [15].

Fully-coherent. All read and write requests from the accelerators are kept coherent with the whole cache hierarchy.

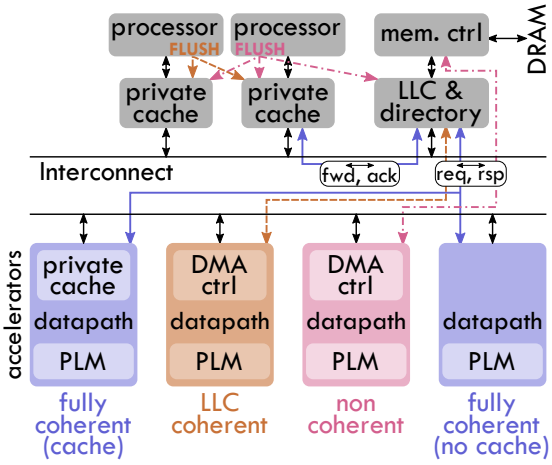

Figure 1: Interaction of the three cache-coherence models with the cache hierarchy in a multi-accelerator SoC.

As Figure 1 depicts, the requests from fully-coherent accelerators are sent to the directory and they may cause a forward request to one or more private caches. This model can be implemented by endowing the accelerators with a private cache [1, 13, 20, 26, 30]. However, the cache is not strictly necessary: for instance, the ARM ACE-lite protocol allows the accelerators to communicate coherently directly on the interconnect, without the need for a private cache [3]. The fully-coherent model is the same model the processor cores use to access memory and it comes with the full overhead of the cache-coherence protocol.

LLC-coherent. The LLC-coherent model avoids all the forward messages to the processor's private caches: the accelerator is only coherent with the LLC, not with the private caches. This model works correctly only if the data requested by the accelerator are not currently cached in a private cache. This can be enforced with a selective flush of the processors' private caches before invoking the accelerator. Normally, the accelerator's task is coarse enough to make the flush overhead negligible. While this model, like the non-coherent one, does not necessarily require the presence of a DMA controller within the accelerator tile, this is a common implementation choice in literature [10].

Cota et al. were among the first to propose this model [11]. Subsequently, we presented a cache-coherence protocol and an architecture to support the LLC-coherent model [16].

The benefit of this approach is to relax the cache-coherence protocol, by leveraging the fact that accelerators have exclusive access to their data structures. At the same time, this model still makes use of the cache hierarchy instead of bypassing it completely, thus improving the execution time and drastically reducing the off-chip accesses.

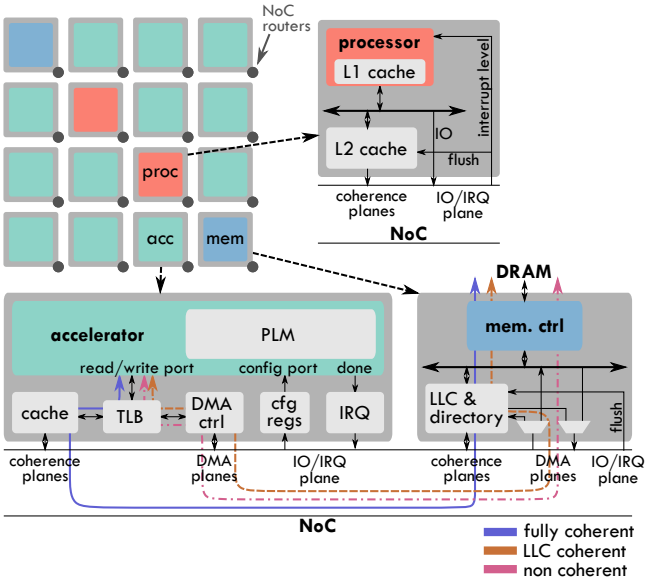

Figure 2: An instance of the ESP architecture with 16 tiles. The tiles interact via a NoC that they access through their sockets. The colored arrows highlight the communication flow for each of the three cache-coherence models as shown in Figure 1.

Non-coherent. Finally, the non-coherent model goes a step further in removing the cache-coherence protocol overhead. It completely bypasses the cache hierarchy by accessing memory directly [9, 11, 22, 26]. Similarly to the LLC-coherent model, the processor's private caches have to be flushed selectively prior to the accelerator's invocation. Additionally, however, also the LLC itself must be flushed because it cannot contain any accelerator's data.

3 ARCHITECTURE

This work is based on the architecture and design methodology of an existing platform for rapid design of heterogeneous SoCs, part of the Embedded Scalable Platforms (ESP) project [8, 21–23]. The latest version of ESP supports all the cache-coherence models for accelerators described in Section 2 [16]. To handle LLC-coherent requests, the cache hierarchy implements a variation of a classic directory-based MESI protocol [28].

ESP is a tiled architecture, where each tile is encapsulated in a configurable socket, which handles the communication mechanisms and some tile-specific services. The socket, for example, takes care of the cache-coherence protocol or in some tiles it implements a DMA engine. All these services are transparent to the content of the tile. This decoupling of the tile from the rest of the system greatly simplifies the integration of heterogeneous components.

Figure 2 shows a four-by-four instance of the ESP architecture. The interconnect is a 2D mesh NoC with six physical planes. As depicted in Figure 2, three planes are devoted to cache-coherence messages, two to DMA transactions and one to memory mapped I/O and interrupts.

ESP features three types of tiles: processor, memory, and accelerator tiles. Thanks to its predisposition toward compositionality, the ESP methodology allows the designer to combine arbitrarily multiple instances of these tiles to compose multi-processor and multi-accelerator SoCs.

Processor tile. The processor tile contains an off-the-shelf processor core, the Leon3 [14], with its write-through L1 caches. An L2 private cache handles the cache-coherence protocol, decoupling the processor core from it. Both the requests to memory mapped I/O and the interrupts are not cached. Instantiating multiple processor tiles yields a multi-core SoC capable of running a full-fledged Linux SMP operating system.

Memory tile. The memory tile grants access to off-chip memory. Each memory tile contains a memory controller, a portion of LLC, and the corresponding part of the MESI protocol directory.

Accelerator tile. The accelerator tile can host any accelerator that complies with a simple latency-insensitive protocol interface [7], which consists of read and write ports, configure-register port, and a completion *done* signal.

To invoke the accelerator, a processor core writes the configuration to the I/O mapped registers of the accelerator's socket. Upon completion of its coarse-grained task, the accelerator emits the *done* signal and the socket sends an interrupt back to the processor. The socket contains a very lightweight translation lookaside buffer (TLB) to map the accelerator's own virtual addresses to physical memory. For this, the socket employs a scatter-gather list, which allows the division of the accelerator's addressable space into large pages in order to have a page table continuous in memory and that fits in the TLB [22]. As a result, the TLB fetches page table entries from memory without the need for operating system support or the need to copy data across memory regions.

The socket of the accelerator tile contains also a simple DMA engine and, optionally, a private cache. This tile supports all the cache-coherence models introduced in Section 2, including both options for the fully-coherent model: cached and uncached. At each accelerator's invocation, one of the configuration registers set by the processor specifies the cache-coherence model of choice. Figure 2 shows the communication flow for each of the models, by matching the colors of Figure 1. Regardless of whether or not the fully-coherent model makes use of the private cache, its messages are routed through the coherence planes of the NoC. The remaining two models communicate with the rest of the cache

hierarchy through DMA transactions. More precisely, the LLC-coherent model sends its DMA requests to the directory, whereas the non-coherent model communicates directly with the memory controller.

Flush implementation. The LLC-coherent and non-coherent models require the flush of part of the cache hierarchy. While ideally we would flush the addresses of the accelerator's dataset only, the Leon3 processor is not capable of selective flushing. Hence, in this implementation of the ESP architecture the caches are flushed in full. However, accelerator related flushes do not evict instructions, but only data.

In the ESP architecture, the accelerator's cache self-flushes at the end of each execution. In this way the flush of the private caches, required by the LLC-coherent and the non-coherent models, needs to be operated only on the processor tiles. This approach leads to a lower number of players in the system actively participating to the cache coherence, because an accelerator's cache is active only if the accelerator is active. Regardless, it is unlikely for an accelerator to execute back-to-back on the same dataset. Therefore, it is good practice to flush out the data right away, so that it can be accessed more easily by the processor or accelerator that needs it next.

Accelerator Communication Patterns. In an ESP architecture, each accelerator is encapsulated within a tile that decouples it from the rest of the system. In fact, from a system viewpoint, the accelerator is fully characterized by its behavior at the socket interface, that is by its communication properties. Since accelerators can differ greatly in the way they communicate, we defined a set of parameters to describe the communication properties of accelerators.

- **Access pattern**: streaming, strided or irregular.
- **Access fraction**: Fraction of the input being accessed. Accelerators with an irregular access pattern seldom access their input dataset in full.
- **Burst length**: Burst length of the memory requests.
- **Stride length**: Stride length of the memory requests. Applies to strided accelerators only.
- **Compute-to-memory ratio**: The ratio of the time spent on computation and total latency of the memory transfers. Typically, an accelerator is either compute bound or memory bound.
- **Reuse factor**: Number of times the accelerator reads or writes the input and output datasets.
- **In-place**: Some accelerator store the outputs in place of the inputs.
- **Input-output ratio**: Size of the input dataset with respect to the output dataset.

We designed the runtime algorithm so that is effective independently of which type of accelerators are in the system.

Listing 1: Main part of the runtime algorithm to select the cache-coherence model of an accelerator.

```
if (new_footprint < PRIVATE_CACHE_SIZE)
  if (active_acc_fully_cnt < MAX_ACC_FULLY_COH)
    coherence = FULLY_COHERENT;
  else
    coherence = LLC_COHERENT;

else if ((active_llc_footprint + new_footprint)
         > LLC_SIZE)
  coherence = NON_COHERENT;

else if (active_acc_llc_cnt >= N_MEM_TILES * 3)
  coherence = NON_COHERENT;

else
  coherence = LLC_COHERENT;
```

4 RUNTIME ALGORITHM

The main motivation for this work comes from our study of the three cache-coherence models for accelerators from a system standpoint [15]. The results of our analysis report no absolute winner among the models: they are each able to outperform the others in some cases. In particular, the optimal model for an accelerator appears to vary at runtime, depending on the system's status and the memory footprint of the accelerators execution. Given these preliminary hints, the primary goal of this work is to understand how to choose at runtime the optimal cache-coherence model for accelerators and to provide an algorithm for doing so.

The ESP architectures support the concurrent execution of accelerators with different cache-coherence models, as well as a runtime selection of the cache-coherence model at each invocation of an accelerator. Here we propose the first algorithm capable of exploiting successfully this feature.

Since, normally, user-space routines offload tasks to accelerators by invoking the accelerator's device driver, our algorithm is executed by the device driver at each accelerator's invocation. The algorithm selects the cache-coherence model that the accelerator should use for this particular invocation. The algorithm takes as input the following runtime variables:

- *new_footprint*: memory footprint of the accelerator to be invoked.
- *active_acc_fully_cnt*: number of active accelerators running with the fully-coherent model.
- *active_acc_llc_cnt*: number of active accelerators using the LLC, i.e. running with the LLC-coherent or the fully-coherent model.
- *active_llc_footprint*: current aggregate memory footprint of all active accelerators using the LLC, i.e. running with the LLC-coherent or the fully-coherent model.

Listing 1 shows the core of the algorithm:

Table 1: Communication properties of the 12 instances of the synthetic accelerator used in the evaluation SoC.

Accelerator ID	1	2	3	4	5	6	7	8	9	10	11	12
Access pattern	stream	stride	stream	irreg	stream	stride	stream	irreg	stream	stride	stream	irreg
Access fraction	1	1	1	1	1	1	1	1/4	1	1	1	1/16
Burst length	64	4	32	4	128	8	64	4	16	4	32	4
Stride length	0	256	0	0	0	32	0	0	0	512	0	0
Compute-mem ratio	1	1	2	4	4	2	8	2	4	4	2	1
Reuse factor	2	4	1	1	4	1	1	4	1	2	4	1
In-place	no	no	yes	yes	no	yes	no	no	yes	no	no	yes
In-out ratio	1	2	4	1	2	4	1	2	4	1	2	4

- If the accelerator's memory footprint is smaller than its private cache, the algorithm chooses the fully coherent model. This is always the case unless the maximum of active accelerators running with the fully-coherent model has been reached. In this case the LLC-coherent is the model of choice. We set the maximum of active fully-coherent accelerators to make sure that the LLC remains inclusive.
- If the aggregate memory footprint of all active accelerators that use the caches, plus the accelerator to be invoked, exceeds the LLC size, then the algorithm selects the non-coherent model. Alternatively, the algorithm chooses the non-coherent model if there is already more than three active accelerators per memory controller that make use of the caches.
- In all the other cases, the algorithm selects the LLC-coherent model.

The rationale of the algorithm is that if the memory footprint of the active accelerators that make use of the cache exceeds the LLC size, then the cache is very likely to incur thrashing. In this case bypassing the caches is preferable, in order to avoid the continuous evictions and loads from main memory. The same idea applies to the accelerator's private cache. Whenever thrashing is not likely, it is preferable to use a model that exploits the caches to save off-chip accesses. Additionally, to avoid congestion, we impose a limit on the maximum number of active accelerators insisting on the LLC.

Remark. Notice how the algorithm uses very little information. The number of memory tiles and the caches sizes are known at design time. Then, a simple global structure maintains the number of active accelerators and their memory footprints. The algorithm could be designed to use also some of the communication properties of the accelerators in the system. However, we intentionally propose an algorithm as simple and generic as possible, so that it is easily applicable to any multi-accelerator heterogeneous SoC.

5 EXPERIMENTAL SETUP

Evaluation SoC. By designing a synthetic accelerator whose communication properties are configurable, we composed an SoC with a wide range of accelerators in terms of their communication properties. We designed the accelerator in synthesizable SystemC and we applied high-level synthesis (HLS) with Cadence Stratus HLS, a commercial tool, to generate the RTL description.

Our evaluation SoC features 12 instances of this synthetic accelerator with a mixed set of communication characteristics as reported in Table 1. At each invocation, each instance of the synthetic accelerator receives as configuration the size of the input dataset and it operates accordingly.

We ran all the experiments on an SoC configured as in Figure 2: two processor tiles, two memory tiles (i.e. two memory controllers) and twelve instances of our synthetic accelerator as presented in Section 5. The size of the Leon3 L1 caches is $16KB$, while all the other private caches in the system are $64KB$ in size. The split LLC measures $1MB$ per partition, that is $2MB$ overall. The bandwidth to off-chip memory is throttled by an AMBA AHB bus to one 32-bit word access per cycle. We deployed the full SoC on a Xilinx Virtex7 FPGA. Since the FPGA operates at lower frequencies than an ASIC, to achieve an off-chip access penalty equivalent to an ASIC implementation, we configured the DDR3 memory to operate at its slowest possible frequency.

Synthetic application. To evaluate our algorithm we constructed an application which is heterogeneous in how it invokes the accelerators. The number of concurrently active accelerators varies considerably and there are data dependencies across invocations. Additionally, the workloads on which the accelerators operate have a wide range of sizes.

As shown in Table 2, our synthetic application goes through a series of phases, where each phase differs from the others in the maximum number of active accelerators and in the size of the workloads. *Small* memory footprints are smaller than the accelerator private cache, whereas *large* ones are larger than the LLC. For each phase, the application spawns as many threads as the maximum of active accelerators.

Table 2: Phases of the synthetic application.

App phases	Memory footprints sizes	Max active accelerators	App threads
1	variable	1	2
2	large	1	2
3	small	1	2
4	variable	6	7
5	large	6	7
6	small	6	7
7	variable	12	13
8	large	12	13
9	small	12	13

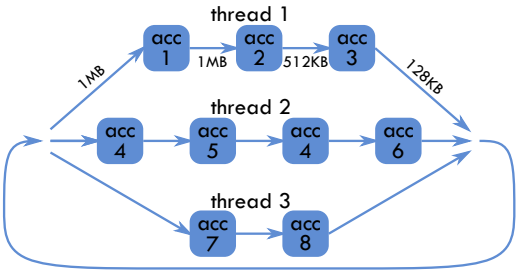

Figure 3: Sample of a phase of the synthetic application. Each square is the invocation of one of eight accelerators. Each forward arrow indicates a data dependency labeled with the memory footprint.

Figure 3 contains an example of what a phase with three threads looks like. Each thread invokes between two to four accelerators that have data dependencies. As the feedback loop indicates, the phase can be executed multiple times on newly allocated data.

6 EXPERIMENTAL RESULTS

We want to prove that, thanks to our simple algorithm, a system with heterogeneous cache-coherence models for accelerators has better performance than an homogeneous one. Therefore, we first run our synthetic application with three baseline configurations, where each baseline works with a single cache-coherence model throughout the whole application. At a minimum, for each phase of the application we would expect the execution with our algorithm to be as good as the best of the three baselines in that phase. Then, whenever the heterogeneity of the coherence models can bring benefits, we expect the algorithm to improve over the best of the homogeneous baselines.

We measure performance in terms of execution time and off-chip memory accesses. We are able to collect exact statistics on DRAM accesses, thanks to the monitor services provided by the ESP architecture [21, 22].

Each graph in Figure 4 reports the amount of DRAM accesses on the y-axis and the logarithm of the execution time in seconds on the x-axis. The DRAM accesses are shown as incremental over time, thus the value of the right-most point of the curve corresponds to the total of the off-chip accesses as well as the execution time. First, the results confirm one of the premises of this paper: none of the coherence models is consistently dominated by the other ones in performance. By looking at the per-phase results, we see that each of the baselines wins at least in one phase.

The execution with our algorithm (*AUTO*) consistently wins in terms of execution time, or in some worst cases it ties. This is the most important result and confirms the efficiency of the proposed algorithm. *AUTO* wins by 100% in best case (i.e. Phase 2), but in other cases simply matches the execution time of the fastest baseline (e.g. Phase 1). This means that *AUTO* matches at least the best homogeneous baseline, and wherever there is the opportunity to exploit the benefits of heterogeneous models it does so. For example, for Phase 1 the memory footprints are so big that the non-coherent model is always the best choice.

Figure 5 shows in detail the execution of Phase 4. The algorithm yields the lowest execution time, with an improvement of 20% over the non-coherent baseline and of approximately 100% over the other two baselines. Here, differently from Figure 4, the DRAM accesses are reported per time sample. So the end of DRAM accesses correspond to the end of application execution.

With respect to external memory accesses, *AUTO* looses in a few cases, but never by more than 10% and in those cases it still wins in terms of execution time. The best win, by 100% takes place in Phase 2. Among the baselines, although the cached models tend to be the most efficient, there is no absolute winner. For instance, if the caches incur thrashing for long periods, the non-coherent model turns out to be at least as good as the others. The savings that our algorithm obtains in terms of number of memory accesses are the results of its ability to chose the non-coherent model when a high degree of cache thrashing may occur.

To assess the performance of our algorithm over the whole application, we report the geometric means of the per-phase speedup and memory accesses ratio of *AUTO* with respect to each of the baselines. The results are listed in Figure 3. Our algorithm achieves speedups between 1.39 and 1.81. Moreover, the off-chip accesses of the application running with our algorithm are between 44% and 71% with respect to the baseline executions.

7 RELATED WORK

The LLC-coherent model has received less attention than the fully-coherent and non-coherent models, which are the main models for loosely-coupled accelerators in the literature [27].

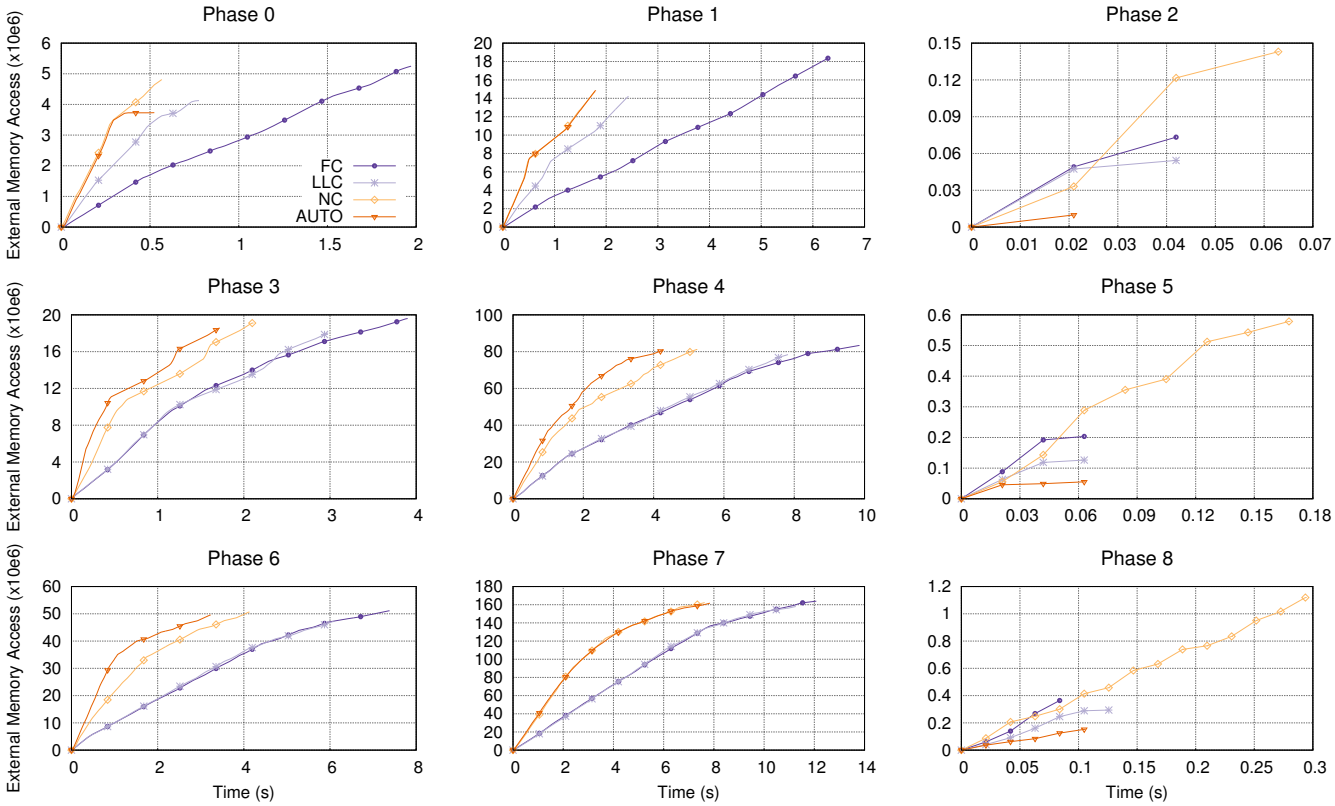

Figure 4: Phase by phase execution time and incremental DRAM accesses of the synthetic app.

Fully-coherent accelerators appear in the industry both as off-chip [24, 29] and on-chip [13] components, but in bus based systems only. Instead, we focus on NoC-based SoCs, which are expected to become the standard for systems with a large number of components [4, 5, 25]. While most approaches endow the accelerators of a private cache, some have the accelerator communicate directly on the bus with a subset of the cache-coherence protocol messages. Once again the existing implementations are bus-based [6, 17]. A similar approach over a NoC would require a costly multi-cast of invalidation and recall messages to the private caches.

Non-coherent accelerators were initially reserved a separate memory space [20], but more recent works, just like ours, apply the shared memory paradigm to avoid copying data across address spaces [9, 18, 22].

We are aware of only a few studies that compare or propose cache-coherence models for accelerators. Kumar et al. presented three variations of the fully-coherent model [19]. Shao et al., instead, evaluated the non-coherent and fully-coherent models [26]. Finally, Cota et al. analyzed LLC-coherent and non-coherent accelerators [11]. These works focus on the simulation of bus-based systems with few accelerators. Instead, we based our research on FPGA prototyping and on a NoC-based SoC with up to 12 accelerators. This allows us to run complex multi-threaded applications, on top of Linux SMP, that invoke multiple accelerators operating on large workloads. The same experimental approach that was used in the two works that constitute the motivation of this paper, as they show the potential benefits of a system with heterogeneous coherence models for accelerators [15, 16].

As SoCs are ever more heterogeneous, the concept of heterogeneous cache-coherence protocols has gained interest. Alsop et al. have proposed Spandex, which can directly and coherently interface devices with different coherence properties and memory demands [1]. Although Spandex has only been evaluated on CPU-GPU systems, it is designed to apply to other components as well, like co-processors or accelerators.

Similarly to *Fusion* [19], other works explored the case of multiple accelerators sharing the same private cache or scratchpad to optimize the memory requirements [9, 20]. Arguably, a group of accelerators sharing the same private cache or PLM can be defined by its aggregate communication pattern and workload size. Our algorithm is potentially beneficial also in this case and it can easily be extended following

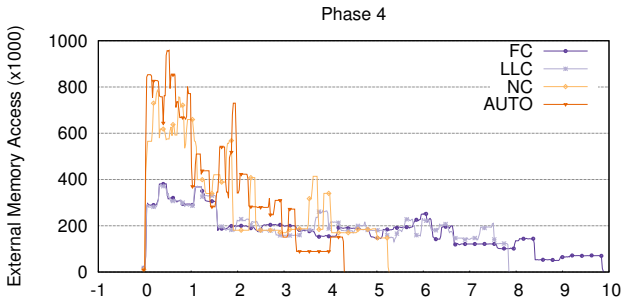

Figure 5: Execution time and DRAM accesses per time sample of phase 4 of the synthetic app.

the same intuition that compares the size of the aggregate workload with that of the caches.

8 CONCLUSION

In the context of multi-accelerator SoCs, we showed how the heterogeneity of cache-coherence models can lead to speedups of at least 40% and how it can reduce the off-chip accesses by a minimum of 30%. For this purpose, we proposed a runtime algorithm capable of selecting the proper cache-coherence model at each accelerator's invocation. The algorithm is lightweight and makes use of information that is general enough to apply easily to any SoC, like the number of active accelerators, the caches capacity, and the size of the accelerators' memory footprints.

ACKNOWLEDGMENTS

This work was supported in part by DARPA (C#: FA8650-18-2-7862). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

Table 3: Execution speedup and reduction of off-chip memory accesses achieved by our algorithm with respect to the three homogeneous baselines.

	Execution time: Baseline / AUTO	DRAM accesses: AUTO / baseline
non-coherent	1.49	0.44
LLC-coherent	1.39	0.71
fully-coherent	1.81	0.58

REFERENCES

[1] J. Alsop, M. D. Sinclair, and S. V. Adve1. 2018. Spandex: A Flexible Interface for Efficient Heterogeneous Coherence. In *Proc. of ISCA*.
[2] Mobileye (an Intel Company). 2018. Towards Autonomous Driving. URL: https://s21.q4cdn.com/600692695/files/doc_presentations/2018/CES-2018-final-MBLY.pdf. CES.
[3] ARM 2017. *AMBA AXI and ACE Protocol Specification.* ARM.
[4] J. Balkind et al. 2016. OpenPiton: An Open Source Manycore Research Framework. In *Proc. of ASPLOS.*
[5] L. Benini and G. De Micheli. 2002. Networks on Chips: A New SoC Paradigm. *IEEE Computer* (2002).
[6] B. Blaner et al. 2013. IBM POWER7+ Processor On-Chip Accelerators for Cryptography and Active Memory Expansion. *IBM J. Research & Development* (2013).
[7] L. P. Carloni. 2015. From Latency-Insensitive Design to Communication-Based System-Level Design. *Proc. of the IEEE* (2015).
[8] L. P. Carloni. 2016. The Case for Embedded Scalable Platforms. In *Proc. of DAC.*
[9] Y. T. Chen et al. 2013. Accelerator-rich CMPs: From Concept to Real Hardware. In *Proc. of ICCD.*
[10] J. Cong et al. 2014. Accelerator-rich Architectures: Opportunities and Progresses. In *Proc. of DAC.*
[11] E. Cota et al. 2015. An Analysis of Accelerator Coupling in Heterogeneous Architectures. In *Proc. of DAC.*
[12] M. Ditty et al. 2014. NVIDIA'S Tegra K1 System-on-Chip. In *Proc. of HCS.*
[13] H. Franke et al. 2010. Introduction to the Wire-Speed Processor and Architecture. *IBM J. Research & Development* (2010).
[14] Jiri Gaisler. 2004. An Open-Source VHDL IP Library with Plug & Play Configuration. *Building the Information Society* (2004).
[15] D. Giri, P. Mantovani, and L. P. Carloni. 2018. Accelerators & Coherence: An SoC Perspective. *IEEE Micro* (2018).
[16] D. Giri, P. Mantovani, and L. P. Carloni. 2018. NoC-Based Support of Heterogeneous Cache-Coherence Models for Accelerators. In *Proc. of NOCS.*
[17] John Goodacre. 2008. *The Effect and Technique of System Coherence in ARM Multicore Technology.* MPSoC.
[18] Y. Hao et al. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *Proc. of HPCA.*
[19] S. Kumar et al. 2015. Fusion: Design Tradeoffs in Coherent Cache Hierarchies for Accelerators. In *Proc. of ISCA.*
[20] M. Lyons et al. 2012. The Accelerator Store: A Shared Memory Framework for Accelerator-based Systems. *TACO* (2012).
[21] P. Mantovani et al. 2016. An FPGA-based Infrastructure for Fine-Grained DVFS Analysis in High-Performance Embedded Systems. In *Proc. of DAC.*
[22] P. Mantovani et al. 2016. Handling Large Data Sets for High-performance Embedded Applications in Heterogeneous Systems-on-chip. In *Proc. of CASES.*
[23] P. Mantovani, G. Di Guglielmo, and L. P. Carloni. 2016. High-level Synthesis of Accelerators in Embedded Scalable Platforms. In *Proc. of ASPDAC.*
[24] S. Neuendorffer and F. Martinez-Vallina. 2013. Building Zynq® Accelerators with Vivado® High-Level Synthesis. In *Proc. of FPGA.*
[25] P. Pande et al. 2005. Performance Evaluation and Design Trade-offs for Network-on-chip Interconnect Architectures. *IEEE Trans. on Computers* (2005).
[26] Y. Shao et al. 2016. Co-designing Accelerators and SoC Interfaces Using gem5-Aladdin. In *Proc. of MICRO.*
[27] Y. Shao and D. Brooks. 2015. *Research Infrastructures for Hardware Accelerators.* Morgan & Claypool.
[28] D. Sorin et al. 2011. *A Primer on Memory Consistency and Cache Coherence.* Morgan & Claypool.
[29] J. Stuecheli. 2013. POWER8. In *Proc. of the IEEE Hot Chips Symp.*
[30] J. Stuecheli et al. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM J. Research & Development* (2015).
[31] Xilinx. 2018. Adaptable Intelligence: The Next Computing Era. Keynote at the 30th Hot Chips Symposium.