Runtime Reconfigurable Memory Hierarchy in Embedded Scalable Platforms

Luca Carloni

Department of Computer Science
Columbia University in the City of New York

24th Asia and South Pacific Design Automation Conference (ASPDAC), January 21-24, 2019
Heterogeneous Architectures Are Emerging Everywhere

[Source: "Xeon+FPGA Tutorial @ ISCA’16"]

[Source: www.xilinx.com/]

[Source: https://cloudplatform.googleblog.com/]

[Source: www.mobileye.com/]

[Source: https://blogs.nvidia.com/]

[Source: https://aws.amazon.com/ec2.instance-types/f1/]

[Source: www.microsoft.com/]

©Luca Carloni – Columbia University
A (Perhaps Easy?) Prediction: No Single Architecture Will Emerge as the Sole Winner

• The migration from homogeneous multi-core architectures to heterogeneous System-on-Chip architectures will accelerate, across almost all computing domains
  – from IoT devices, embedded systems and mobile devices to data centers and supercomputers specialization will be the key to realize competitive systems

• A heterogeneous SoC will combine an increasingly diverse set of components
  – different CPUs, GPUs, hardware accelerators, memory hierarchies, I/O peripherals, sensors, reconfigurable engines, analog blocks...

• The set of heterogeneous SoCs in production in any given year will be itself heterogeneous!
  – no single SoC architecture will dominate all the markets
Where the Key Challenges in SoC Design Are…

• The biggest challenges are (and will increasingly be) found in the complexity of system integration
  – How to design, program and validate scalable systems that combine a very large number of heterogeneous components to provide a solution that is specialized for a target class of applications?

• How to handle this complexity?
  – raise the level of abstraction to System-Level Design
  – adopt compositional design methods with the Protocol & Shell Paradigm
  – promote Design Reuse
Embedded Scalable Platforms (ESP)

- The flexible architecture simplifies the integration of heterogeneous components by
  - balancing regularity and specialization
  - relying on the Protocol & Shell paradigm and scalable communication infrastructure
- The system-level design methodology promotes HW/SW co-design and is supported by
  - a mix of commercial and in-house CAD tools
  - a growing library of reusable IP blocks

[ L. P. Carloni, The Case for Embedded Scalable Platforms, DAC 2016 ]
The ESP Scalable Architecture Template

Template Properties

- **Regularity**
  - tile-based design
  - pre-designed on-chip infrastructure for communication and resource management

- **Flexibility**
  - each ESP design is the result of a configurable mix of programmable tiles and accelerator tiles

- **Specialization**
  - with automatic high-level synthesis of accelerators for key computational kernels

**Processor Tiles**
- each hosting at least one configurable processor core capable of running an OS

**Accelerator Tiles**
- synthesized from high-level specs

**Other Tiles**
- memory interfaces, I/O, etc.

**Network-on-Chip (NoC)**
- playing key roles at both design and run time
Our System-Level Design Approach: Key Ingredients

- **Develop Platforms, not just Architectures**
  - A platform combines an architecture and a companion design methodology

- **Raise the level of abstraction**
  - Move from RTL Design to **System-Level Design**
  - Move from ISA simulators to **Virtual Platforms**
  - Move from Verilog/VHDL to **SystemC**, also an IEEE standard
  - Move from Logic Synthesis to **High-Level Synthesis** (both commercial and in-house tools), which is the key to enabling rich design-space exploration

- **Adopt compositional design methods**
  - Rely on customizable libraries of HW/SW interfaces to simplify the integration of heterogeneous components

- **Use formal metrics for design reuse**
  - Synthesize Pareto frontiers of optimal implementations from high-level specs

- **Build real prototypes (both chips and FPGA-based full-system designs)**
  - Prototypes drive research in systems, architectures, software and CAD tools

©Luca Carloni – Columbia University
Example of an ESP Based-Design: FPGA Prototype to Accelerate Wide-Area Motion Imagery

- **Design:** Complete design of WAMI-App running on an FPGA implementation of an ESP architecture
  - featuring 1 embedded processor, 12 accelerators, 1 five-plane NoC, and 2 DRAM controllers
  - SW application running on top of Linux while leveraging multi-threading library to program the accelerators and control their concurrent, pipelined execution
  - Five-plane, 2D-mesh NoC efficiently supports multiple independent frequency domains and a variety of platform services

[P. Mantovani, L. P. Carloni et al., *An FPGA-Based Infrastructure for Fine-Grained DVFS Analysis in High-Performance Embedded Systems*, DAC 2016]
How to Couple Accelerators, Processors and Memories?

- There are two main models of coupling accelerators with processors, memories
  - Tightly-Coupled Accelerators
    - designed with the processor core
    - located within the processor core
    - execute fine-grain tasks on small datasets
    - typically accessed via specialized instructions
  - Loosely-Coupled Accelerators
    - designed independently from the processor core
    - located outside the processor core
    - execute coarse-grain tasks on large datasets
    - typically accessed via device drivers

The accelerator model enables the definition of a configurable interface that simplifies the integration of the accelerator within any ESP instance by decoupling the design of any accelerators from the design of the rest of the SoC.

- The behavior of loosely-coupled accelerators has 4 main phases:
  - configuration, input, compute, output

- I/O phases transfer chunks of data from DRAM to the PLM:
  - these transfers are specified with TLM primitives, implemented with DMA mechanisms

- The accelerator model enables the definition of a configurable interface that simplifies the integration of the accelerator within any ESP instance:
  - by decoupling the design of any accelerators from the design of the rest of the SoC
Loosely-Coupled Accelerators

• Major speedups and energy savings:
  – highly parallel and customized datapath
  – aggressively banked, multi-ported, private local memory (PLM)

• What should the cache coherence model for accelerators be?
  – 3 main models in literature
    [D. Giri et al., IEEE Micro ‘18]
Accelerator Models: Fully Coherent

- Coherent with the entire cache hierarchy
  - same coherence model as the processor

- Programming requirements
  - race-free accelerator execution

- Implementation variants
  - generally bus-based
  - accelerators may own a cache
    - IBM CAPI, [Y. Shao et al., MICRO ‘16], [M. J. Lyons et al., TACO ‘12]
    - ARM ACE-lite

© Luca Carloni
Accelerator Models: Non Coherent

- Not coherent with cache hierarchy
  - caches are by-passed while talking with DRAM

- Programming requirements
  - race-free accelerator execution
  - flush all caches prior to accelerator execution

- Implementation variants
  - generally NoC-based & DMA-based
    - [Y. Chen et al., ICCD ‘13]
    - [E. Cota et al., DAC ‘15]
    - [Y. Shao et al., MICRO ‘16]
Accelerator Models: LLC Coherent

- Coherent with LLC only
  - processors’ private caches are by-passed while talking with the LLC

- Programming requirements
  - race-free accelerator execution
  - flush processors’ private caches prior to accelerator execution

- Implementation variants
  - first proposed by [E. Cota et al., DAC ‘15]
  - only 1 implementation in literature [D. Giri et al., NOCS ‘18]
Motivation: Why Different Coherence Models?

- The best choice of coherence model varies at runtime with the accelerator workload size and with the number of active accelerators.
- LLC-coherent and fully-coherent models can significantly reduce the number of off-chip memory accesses.

**RULE OF THUMB**

- **BEST MODEL**
  - fully-coherent model
  - LLC-coherent model
  - non-coherent model

  private cache size  |  LLC size  |  ~ memory footprint of workload

[©Luca Carloni](https://doi.org/10.1109/MM.2018.1)

Heterogeneous Coherence: Experimental Setup

- **FFT1D**
  - streaming memory access

- **Sort**
  - no temporal locality, but in-place (i.e. in the PLM) data processing

- **FFT2D**
  - streaming memory access, but two phases with sequential dependency

- **SPMV**
  - asymmetric data reuse with irregular access pattern
  - very low compute-to-memory ratio

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
Results: Comparing the Speedup of Non-Coherent vs. LLC-Coherent Accelerators (Running *Standalone*)

- Compared to non-coherent accelerators, the relative speedup of LLC-coherent accelerators ranges between 0.5x and 4x
  - the memory access count, instead, ranges from 0 to at most 2x (in worst-case scenario)
- Confirmation of the benefits of runtime model selection based on footprint
Contributions

• We propose a **runtime algorithm** to adaptively manage the cache coherence of accelerators
  – we show how to leverage the heterogeneity of cache-coherence models to improve the overall system performance.

• We evaluate the algorithm with:
  – our FPGA-based platform for rapid SoC prototyping, which is part of the Embedded Scalable Platform project
  – synthetic accelerators with a wide range of communication properties
  – synthetic application
    • varying number of concurrently active accelerators
    • variable memory footprint of the accelerators’ workload
Our SoC Platform

- Our design is based on an instance of Embedded Scalable Platforms (ESP)
  - socketed tiles
  - multi-plane NoC
  - easy integration and reuse of heterogeneous components
  - capable of running multi-processor and multi-accelerator applications on Linux SMP
  - support for all three cache-coherence models for accelerators
Processor Tile

- **Main components**
  - single-core processor tiles, with private L2 cache

- **In this work**
  - up to 2 processor tiles
  - 64KB private caches
  - off-the-shelf processor with L1 write-through caches: Leon3
Memory Tile

• **Main components**
  – memory controller
  – LLC and directory
    • can be split over multiple tiles

• **In this work**
  – 2 memory tiles
  – 2MB aggregate LLC (1MB per tile)
Accelerator Tile

• Main components
  – any accelerator complying with a simple interface
  – a small TLB
  – a DMA controller and/or a private cache

• Support for run-time selection of coherence model
  – selection granularity: possible at each accelerator invocation
  – selection method: one I/O write to the configuration registers

©Luca Carloni
The Proposed Algorithm for Adaptive Management of Accelerator Coherence

• Executed by the device driver at each accelerator’s invocation
• Selects the cache-coherence model for the accelerator
• Static inputs: 4
• Dynamic inputs: 4

```c
if (footprint < PRIVATE_CACHE_SIZE)
    coherence = FULLY_COHERENT;
else
    coherence = LLC_COHERENT;
else if ((current_llc_footprint + footprint) > LLC_SIZE)
    coherence = NON_COHERENT;
else if (n_acc_on_llc_or_fully_coherent >= N_MEM_TILES * MAX_ACC_PER_LLC)
    coherence = NON_COHERENT;
else
    coherence = LLC_COHERENT;
```
Synthetic Accelerators

• An accelerator is characterized by its communication properties
  – we defined 8 parameters to describe the communication properties
  – we designed a “master accelerator” with parametrizable communication properties
  – we generated 12 accelerators with a wide range of communication

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

- Application with multiple phases
  - variable memory footprints of the accelerators’ workloads
  - variable number of concurrently active accelerators

Sample of a possible app phase

<table>
<thead>
<tr>
<th>App phases</th>
<th>Memory footprints sizes</th>
<th>Max active accelerators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>variable</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>large</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>small</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>variable</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>large</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>small</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>variable</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>large</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>small</td>
<td>12</td>
</tr>
</tbody>
</table>
Evaluation SoC

- ESP’s GUI
  - the CAD flow from GUI to FPGA bitstream is fully automated

- We deployed this SoC on FPGA and we executed the synthetic application on Linux SMP
Results

- Our algorithm reduces:
  - the execution time by at least 40%
  - the off-chip accesses by at least 30%
Conclusions

• We showed how to exploit the heterogeneity of cache-coherence models
  – We proposed a runtime algorithm to select the proper cache-coherence model at each accelerator’s invocation

• Heterogeneity of cache-coherence models for accelerators can:
  – lead to speedups of at least 40%
  – reduce the off-chip accesses by a minimum of 30%

• The algorithm is general enough to apply to any SoC
  – its inputs are: number of active accelerators, caches capacity, memory footprint of the accelerator workloads
Some Recent Publications