Cache for HLS

A multi-process architecture

Brignone Giovanni

Politecnico di Torino

Politecnico
di Torino

June 7, 2021

Outline

1 Introduction
- Motivation

2 Architecture
- Inlined architecture
- Multi-process architecture

3 Implementation
- Vitis HLS
- Internal architecture

4 Future work

Introduction

Introduction

Motivation

Motivation

- **Problem**:
  big arrays are mapped to DRAM, therefore they can be performance bottlenecks
- **Proposed solution**:
  cache module storing data to fast BRAMs to be integrated into any design with minimal effort

Architecture

Architecture

Inlined architecture

Inlined architecture (Ma Liang)

Accesses to cached array replaced with whole cache logic:

- straightforward implementation
- cache logic mixed with application logic
- single-port only

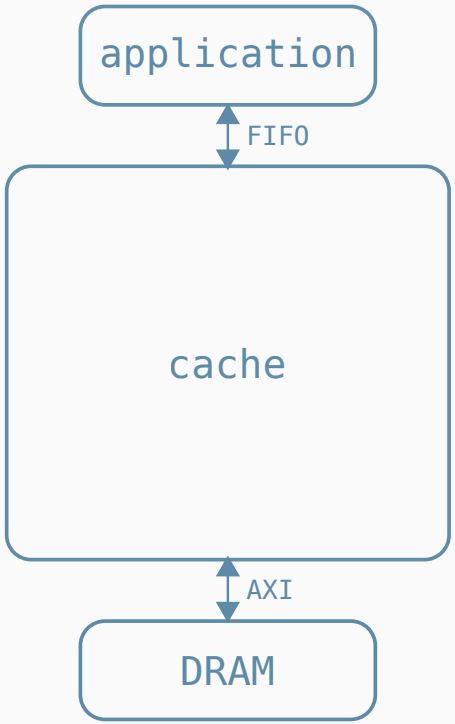
Architecture

Multi-process architecture

Multi-process architecture

Accesses to cached array replaced with request sending and response reception:

- decoupling between application and cache logic (communication through FIFOs only)
- may support multiple ports (work in progress)



Implementation

Implementation

Vitis HLS

Vitis HLS (2020.2)

- **Relevant features**:
  - multiple processes modeling:
    - SW: `std::thread`
    - HW: `DATAFLOW` with *start propagation* disabled
  - `hls::stream` as FIFO implementation
  - loop pipelining
  - automatic port widening
- **Limitations**:
  - automatic `class` disaggregation prevents "`[]` operator" override for `set` operation (pointer to `class` is required)

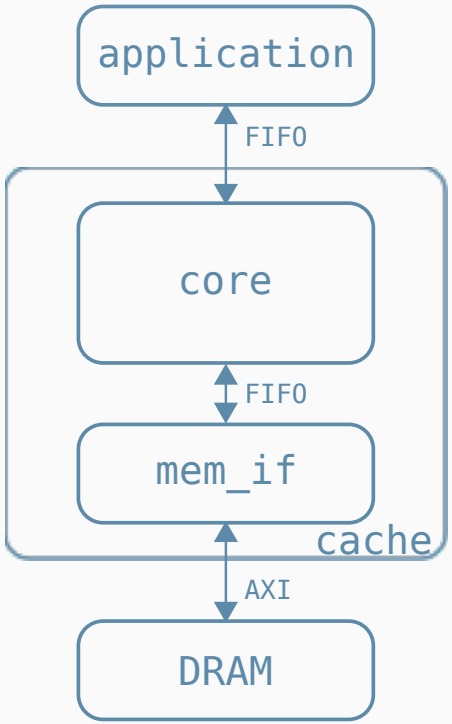
Implementation

Internal architecture

Internal architecture

- **Issue**: persuade HLS to schedule response writing to FIFO in case of HIT early in the pipeline
- **Solution**: split the cache into two processes:
  1. `core`:
     - manage requests from `application`
     - keep cache data structures up-to-date
  2. `mem_if`:
     - manage DRAM accesses

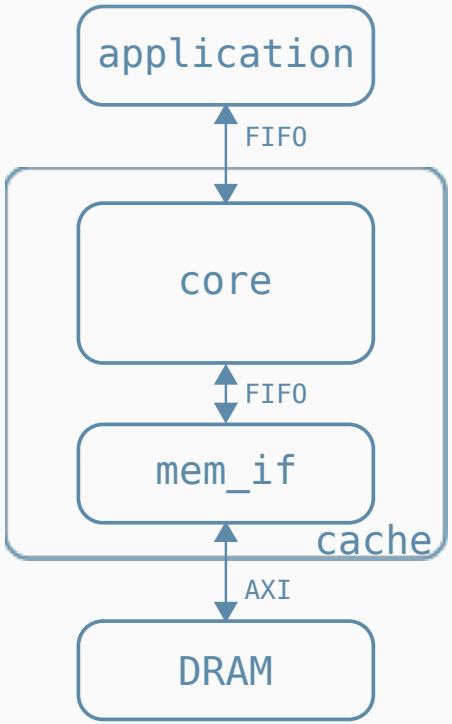
  $\Rightarrow$ synthesizer job is simplified:
  if HIT, response is written in 1 cycle



mem_if IMPLEMENTATION

Manage DRAM accesses:

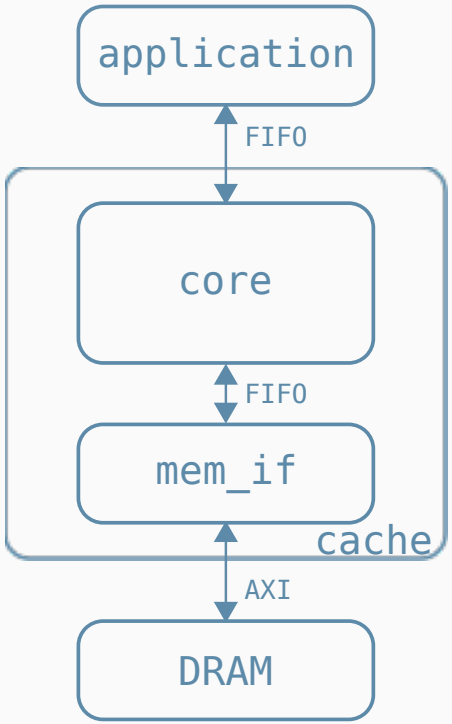
1. read request from `core`
2. access DRAM
3. write response to `core` (if `read` request)



core IMPLEMENTATION

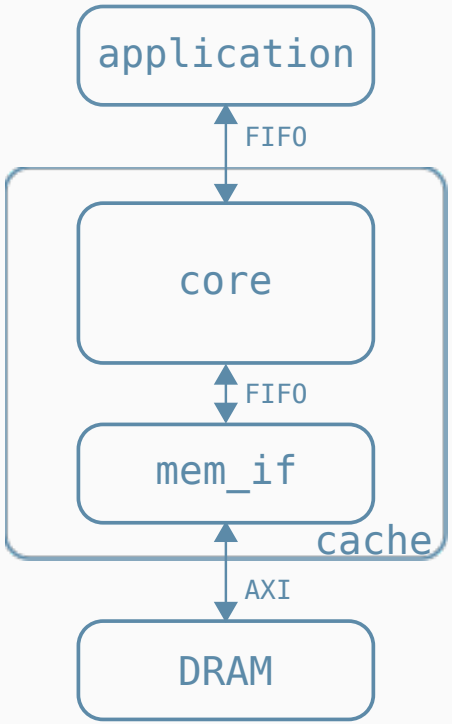
Infinite pipelined loop performing following steps:

1. read request from `application`
2. check if it is an HIT or a MISS
3. if MISS:
   - if the cache line to be overwritten $\texttt{line}_{old}$ has been modified, issue a `write` request of $\texttt{line}_{old}$ to `mem_if`
   - issue a `read` request of the requested line to `mem_if`
   - read `mem_if` response and update BRAM
4. access BRAM
5. write response to `application` (if `read` request)



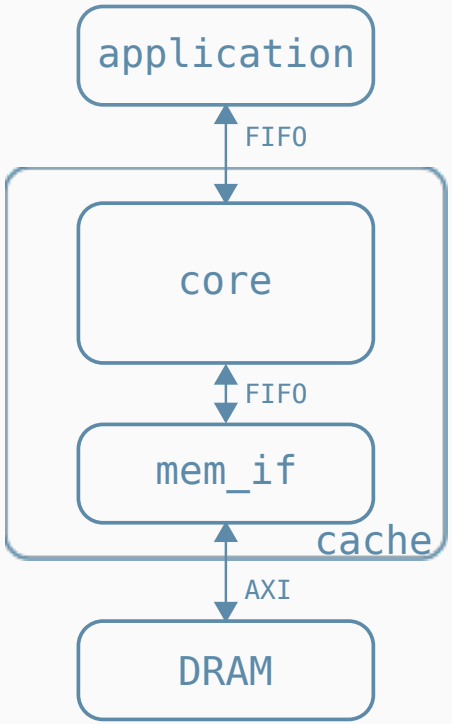
`core` IMPLEMENTATION

- **Problem**: reading BRAM immediately after it is written by the `fill` process causes an increase of II
- **Solution**: store the `mem_if` response in a buffer which can be immediately accessed and update BRAM afterwards



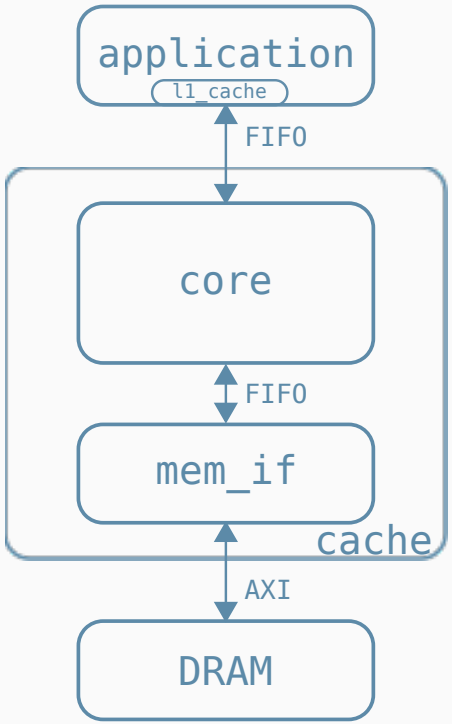
`core` IMPLEMENTATION

- **Problem**: reading BRAM immediately after it is written by a previous `write` request causes an increase of II
- **Solution**: insert a `raw_cache` (single-line cache storing last written line): in case of a `read` request right after a `write` request to the same line, the access is performed to the `raw_cache` instead of BRAM



Request optimization

- **Scenario**:
  - each FIFO access costs 1 cycle
  - accesses to arrays are often sequential
- **Proposed solution**:
  insert a `l1_cache` in the interface, on `application` side: a `read` request gets the whole cache line, stores it and returns the required element; if subsequent requests read the same line, the interface can immediately return data, avoiding sending any request and waiting for response



Future work