Cache for HLS

A multi-process architecture

Brignone Giovanni

June 7, 2021

Politecnico di Torino

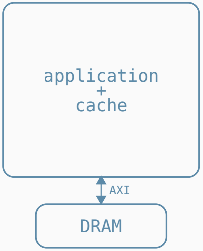
Motivation

- **Problem**:
  big arrays are mapped to DRAM $\Rightarrow$ performance bottlenecks
- **Proposed solution**:
  cache module which should:
  - store data to BRAMs
  - require minimal effort to be integrated into any HLS design

Inlined architecture (Ma Liang)

Array access → Inlined cache logic

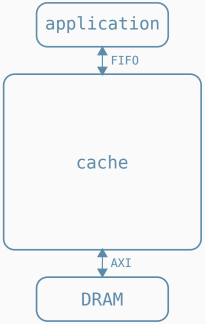
- straightforward architecture
- cache logic mixed with application logic
- single-port only



Multi-process architecture

Array access → Request to separate module

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



Tools

`C++` code compatible with *Vitis HLS* (2020.2).

Tools

`C++` code compatible with *Vitis HLS* (2020.2).

- **Multiple processes modeling**:
  infinite loops parallelized by:
  - SW simulation: `std::thread`
  - Synthesis: `DATAFLOW` with *start propagation* disabled

Tools

`C++` code compatible with *Vitis HLS* (2020.2).

- **Multiple processes modeling**:
  infinite loops parallelized by:
  - SW simulation: `std::thread`
  - Synthesis: `DATAFLOW` with *start propagation* disabled
- **Inter-process communication**: `hls::stream` FIFOs

Tools

`C++` code compatible with *Vitis HLS* (2020.2).

- **Multiple processes modeling**:
  infinite loops parallelized by:
  - SW simulation: `std::thread`
  - Synthesis: `DATAFLOW` with *start propagation* disabled
- **Inter-process communication**: `hls::stream` FIFOs
- **Performance optimization**:
  - loop pipelining: new request each cycle
  - automatic port widening: access one line at a time in DRAM

Tools

`C++` code compatible with *Vitis HLS* (2020.2).

- **Multiple processes modeling**:
  infinite loops parallelized by:
  - SW simulation: `std::thread`
  - Synthesis: `DATAFLOW` with *start propagation* disabled
- **Inter-process communication**: `hls::stream` FIFOs
- **Performance optimization**:
  - loop pipelining: new request each cycle
  - automatic port widening: access one line at a time in DRAM
- **Customization:** cache characteristics set through `template`

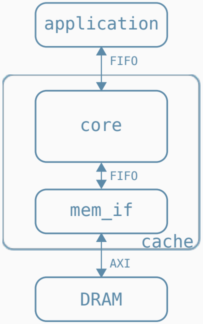
Tools

`C++` code compatible with *Vitis HLS* (2020.2).

- **Multiple processes modeling**:
  infinite loops parallelized by:
  - SW simulation: `std::thread`
  - Synthesis: `DATAFLOW` with *start propagation* disabled
- **Inter-process communication**: `hls::stream` FIFOs
- **Performance optimization**:
  - loop pipelining: new request each cycle
  - automatic port widening: access one line at a time in DRAM
- **Customization:** cache characteristics set through `template`
- **Limitations**: cannot override "`[] operator`" for `set` due to automatic `class` disaggregation

Internal architecture

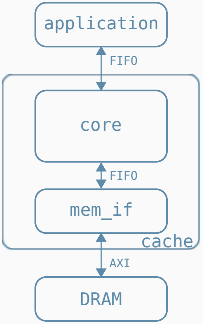
- **Problem**: persuade HLS to schedule response writing to FIFO in case of HIT early in the pipeline



Internal architecture

- **Problem**: persuade HLS to schedule response writing to FIFO in case of HIT early in the pipeline
- **Proposed 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, request is managed in 1 cycle

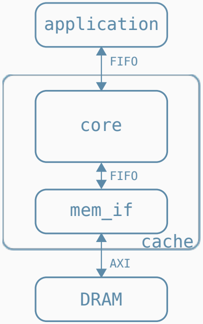


mem_if implementation

Functionality

Manage DRAM accesses:

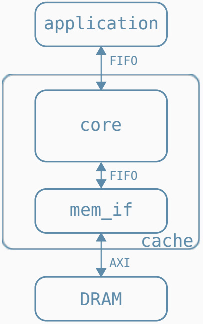
1. read request from core
2. access DRAM
3. write response to core (if read request)



core implementation

Functionality

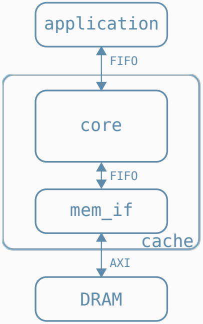
1. read request from application



core implementation

Functionality

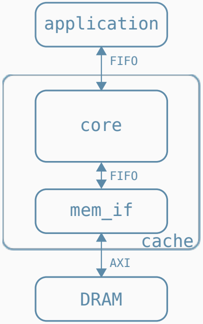
1. read request from `application`
2. check if it is an HIT or a MISS



core implementation

Functionality

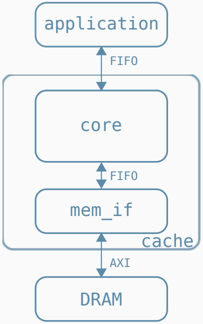
1. read request from `application`
2. check if it is an HIT or a MISS
3. if MISS:



core implementation

Functionality

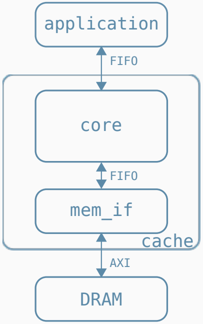
1. read request from application
2. check if it is an HIT or a MISS
3. if MISS:
   - if the cache line to be overwritten $\texttt{line}_{\text{old}}$ has been modified, issue a write request of $\texttt{line}_{\text{old}}$ to mem_if



core implementation

Functionality

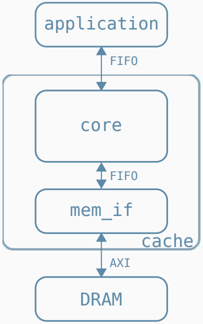
1. read request from `application`
2. check if it is an HIT or a MISS
3. if MISS:
   - if the cache line to be overwritten $\texttt{line}_{\text{old}}$ has been modified, issue a `write` request of $\texttt{line}_{\text{old}}$ to `mem_if`
   - issue a `read` request of the requested line to `mem_if`



core implementation

Functionality

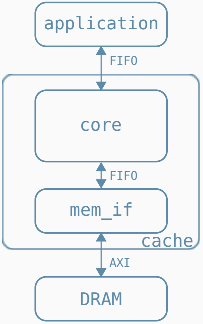
1. read request from `application`
2. check if it is an HIT or a MISS
3. if MISS:
   - if the cache line to be overwritten $\texttt{line}_{\text{old}}$ has been modified, issue a `write` request of $\texttt{line}_{\text{old}}$ to `mem_if`
   - issue a `read` request of the requested line to `mem_if`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



core implementation

Functionality

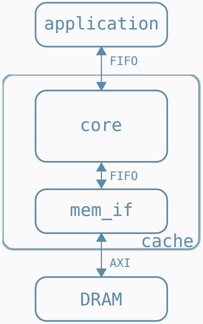
1. read request from `application`
2. check if it is an HIT or a MISS
3. if MISS:
   - if the cache line to be overwritten `line`$_{old}$ has been modified, issue a `write` request of `line`$_{old}$ to `mem_if`
   - issue a `read` request of the requested line to `mem_if`
   - read `mem_if` response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



core implementation

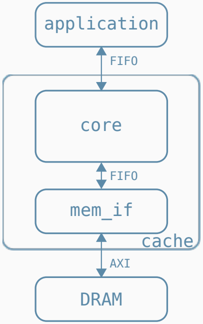
Functionality

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 `line`$_{\text{old}}$ has been modified, issue a `write` request of `line`$_{\text{old}}$ 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`)



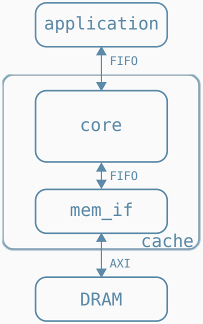
BRAM dependencies - RAW `fill`

- **Problem**: reading BRAM immediately after it is written by the `fill` process causes an increase of II



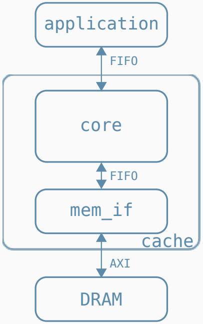
BRAM dependencies - RAW `fill`

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



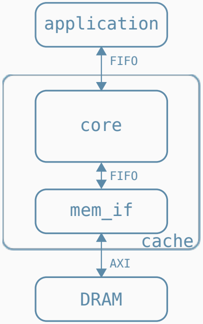
BRAM dependencies - RAW `request`

- **Problem**: reading BRAM immediately after it is written by a previous `write` request causes an increase of II



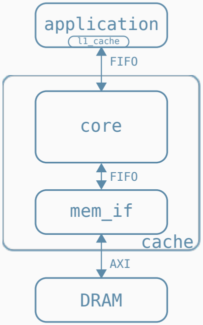
BRAM dependencies - RAW `request`

- **Problem**: reading BRAM immediately after it is written by a previous `write` request causes an increase of II
- **Proposed solution**: `raw_cache` inside `cache`:
  - `write` request: store the written line to a buffer
  - subsequent `read` request to the same line: access the buffer instead of BRAM



Request optimization

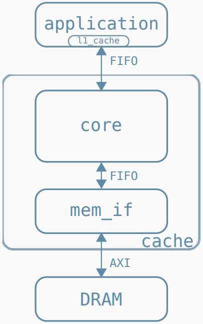
- **Problem**:
  - each FIFO access costs 1 cycle
  - accesses to arrays are often sequential



Request optimization

- **Problem**:
  - each FIFO access costs 1 cycle
  - accesses to arrays are often sequential
- **Proposed solution**: `l1_cache` in the interface (`application` side):
  - `read` request: get the whole cache line, store it in a buffer and return the requested element
  - subsequent `read` request to the same line: access the buffer

$\Rightarrow$ avoid passing through FIFOs



Performance

- **Loop pipelining**:
  both `core` and `mem_if` loops pipelined with `II=1`
- **L1 HIT**: latency of 1 cycle
- **HIT**: latency of 6 cycles

Multi-port cache

- **Problem**: cache manages one request per cycle $\Rightarrow$ unrolling a loop which uses the cache is not trivial

Multi-port cache

- **Problem**: cache manages one request per cycle ⇒ unrolling a loop which uses the cache is not trivial
- **Possible workarounds**: get multiple data elements for each request and do all the unroll in `application`; two possible ways:
  - `get_line`: return a whole cache line
  - pack multiple data elements in a single cache element (e.g. `hls::vector`)

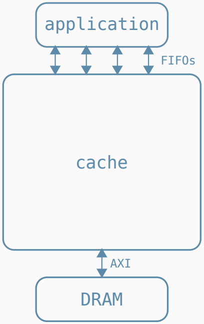
Multi-port cache

- **Problem**: cache manages one request per cycle $\Rightarrow$ unrolling a loop which uses the cache is not trivial
- **Possible workarounds**: get multiple data elements for each request and do all the unroll in `application`; two possible ways:
    - `get_line`: return a whole cache line
    - pack multiple data elements in a single cache element (e.g. `hls::vector`)
- **Proposed solution**: provide N ports, where N is the unroll factor

Multi-port cache

Implementation

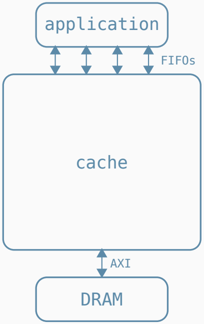
- **Straightforward implementation**:
  - provide N FIFOs on the interface
  - unroll core loop by a factor N



Multi-port cache

Implementation

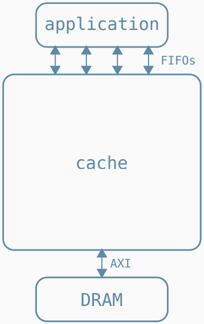
- **Straightforward implementation**:
  - provide N FIFOs on the interface
  - unroll `core` loop by a factor N
- **Problem**: all the N requests may refer to the same line $\rightarrow$ each iteration must wait completion of previous one



Multi-port cache

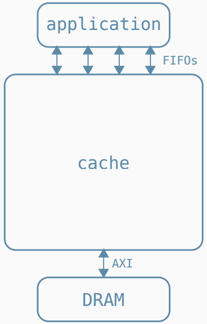
Implementation

- **Proposed solution**:
  1. read all the N requests
  2. check if they are all HIT
  3. if at least one MISS issue all the requests to `mem_if`
  4. access BRAM
  5. write all the responses



Multi-port cache
Implementation

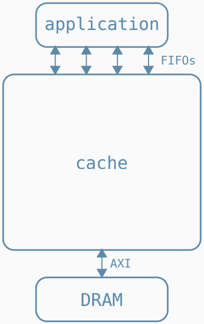
- **Proposed solution**:
  1. read all the N requests
  2. check if they are all HIT
  3. if at least one MISS issue all the requests to `mem_if`
  4. access BRAM
  5. write all the responses



Multi-port cache

Implementation

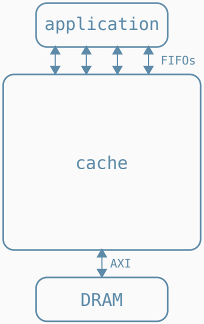
- **Proposed solution**:
  1. read all the N requests
  2. check if they are all HIT
  3. if at least one MISS issue all the requests to `mem_if`
  4. access BRAM
  5. write all the responses
  - cache addressing mode converted to an associative one



Multi-port cache

Implementation

- **Proposed solution**:
  1. read all the *N* requests
  2. check if they are all HIT
  3. if at least one MISS issue all the requests to `mem_if`
  4. access BRAM
  5. write all the responses
  - cache addressing mode converted to an associative one
  - possibly implement `core` at RTL to have full control on scheduling


application
FIFOs
cache
AXI
DRAM