POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Acceleration by Separate-Process Cache for Memory-Intensive Algorithms on FPGA via High-Level Synthesis

Supervisor
Prof. Luciano Lavagno

Candidate
Giovanni Brignone
ID: 274148

Academic year 2020-2021

Abstract

The end of the Moore's Law validity is making the performance advance of Software run on general purpose processors more challenging than ever. Since current technology cannot scale anymore it is necessary to approach the problem from a different point of view: application-specific Hardware can provide higher performance and lower power consumption, while requiring higher design efforts and higher deployment costs.

The problem of the high design efforts can be mitigated by the High-Level Synthesis (HLS), since it helps improve designer productivity thanks to convenient Software-like tools.

The problem of high deployment costs can be tackled with FPGAs, which allow implementing special-purpose Hardware modules on general-purpose underlying physical architectures.

One of the open issues of HLS is the memory bandwidth bottleneck which limits performance, especially critical in case of memory-bound algorithms.

FPGAs memory system is composed of three main kinds of resources: registers, Block RAMs (BRAMs) and external Dynamic RAMs (DRAMs). Current HLS tools allow to exploit this memory hierarchy manually, in a scratchpad-like fashion: the objective of this thesis work is to automate the memory management by providing an easily integrable and fully customizable cache system for HLS.

The proposed implementation has been developed using Vitis™HLS tool by Xilinx Inc..

The first development phase produced a single-port cache module, in the form of a C++ class configurable through templates in terms of number of sets, ways, words per line and replacement policy. The cache lines have been mapped to BRAMs. To obtain the desired performance, an unconventional (for HLS) multiprocess architecture has been developed: the cache module is a separate process with respect to the algorithm using it: the algorithm logic sends a memory access request to the cache and reads its response, communicating through FIFOs.

In the second development phase, the focus was put on performance optimization, in two dimensions: increasing the memory hierarchy depth by introducing a Level 1 (L1) cache and increasing parallelism by enabling multiple ports.

The L1 cache is composed of cache logic inlined in the user algorithm: this solution allows to cut the costs of FIFOs communications. To keep L1 cache simple it has been implemented with a write-through write policy, therefore it provides advantages for read accesses only. It is configurable in the number of lines and each line contains the same number of words of the associated Level 2 (L2) cache.

The multi-port solution provides a single L2 cache accessible from multiple FIFOs ports, each of which can be associated with a dedicated L1 cache. It is possible to specify the number of ports through a template parameter and it typically corresponds to the unrolling factor of the loop in which the cache is accessed.

In order to evaluate performance and resource usage impact of the developed cache module, multiple algorithms with different memory access patterns have been synthesized and simulated, with all data accessed to DRAM (performance lower bound), to BRAM (performance higher bound) and to cache (with multiple configurations).

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

AXI Advanced eXtensible Interface

BRAM Block RAM

DRAM Dynamic RAM

FIFO First-In First-Out

FPGA Field-Programmable Gate Array

FSM Finite-State Machine

HDL Hardware Description Language

HLS High-Level Synthesis

HW Hardware

II Initiation Interval

IPC Inter-Process Communication

L1 Level 1

L2 Level 2

LRU Least Recently Used

LSB Least Significant Bit

MSB Most Significant Bit

RAM Random Access Memory

RAW Read After Write

RTL Register-Transfer Level

SW Software

1 Background

The literature about cache systems, the High-Level Synthesis state of the art and an analysis of the resources available on board modern FPGAs are the fundamental background for this thesis work.

1.1 Cache

Memory devices are crucial components of computing systems as they can pose an higher bound in terms of performance, especially when executing memory-intensive algorithms. The ideal memory should be fast, large and cheap, but current technology forces the designer to choose a trade-off between the metrics.

A common solution to this problem is to set up a memory hierarchy in which fast but small memories are paired with large but slow memories, which allows getting good performance on average while containing costs.

This hierarchy can be managed by two main approaches:

- *Scratchpad*: different memories belongs to different addressing spaces: the user is in charge of manually choosing what memory to access: this approach allows to optimally exploit the hierarchy at the cost of high design effort.
- *Cache*: different memories belongs to the same addressing space: the system automatically uses the whole hierarchy, exploiting spatial locality (accessed data is likely physically close to previously accessed data) and temporal locality (accessed data has likely recently been accessed), which are typical of many algorithms.

1.1.1 Structure

A cache memory is logically split into *sets* containing *lines* (or *ways*) which are in turn made up of *words*, as shown in Figure 1.1.

Whenever a word w is requested, there are two possibilities:

- *Hit*: w is present in the cache: the request can be immediately fulfilled.
- *Miss*: w is not present in the cache: it is necessary to retrieve it from lower level memory before fulfilling the request.

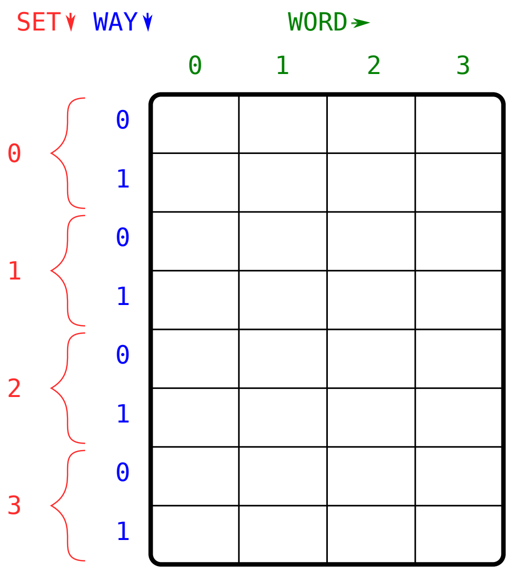


Figure 1.1: Cache logic structure.

During the data retrieving, a cache line is filled with a block of contiguous words loaded from the lower level memory, trying to exploit spatial locality of future accesses, while mapping policies and replacement policies determine which cache line to overwrite, trying to exploit temporal locality.

If the cache memory is writable, data consistency is ensured by a consistency policy.

1.1.2 Policies

Mapping policy

The mapping policy is in charge of statically associating a lower level memory line to a cache set.

The *set associative* policy is the most common mapping policy: given a cache memory with s sets of w words, the word address (referred to the lower level memory) bits are split into three parts (as shown in Figure 1.2):

1. $\log_2(w)$: offset of the word in the line.
2. $\log_2(s)$: set.
3. Remaining MSBs: tag identifying the specific line.

Special cases of this policy are:

- *Direct mapped* policy: each set is composed of a single line: the set bits identify a specific cache line, therefore there is no need for a replacement policy.

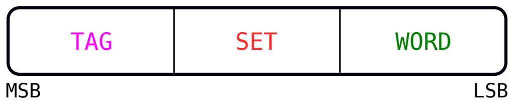


Figure 1.2: Set associative policy address bits meaning.

- *Fully associative* policy: there is only a single set, therefore the line is fully determined by the replacement policy.

Replacement policy

The replacement policy is in charge of dynamically associating a lower level memory line to a cache line of a set.

Multiple solutions of this problem have been developed, trying to maximize the temporal locality exploitation. Among the most commonly used solutions there are:

- *First-In First-Out*: the line to be replaced is the first one that has been inserted to the cache.
- *Least Recently Used*: the line to be replaced is the one that has least recently been accessed.

Consistency policy

The consistency policy is in charge of ensuring data consistency between memories belonging to different hierarchy levels.

The most common solutions to this problem are:

- *Write-back*: write accesses are performed to the highest level memory and lower level memories are updated when the cache line is replaced only.
- *Write-through*: each write access is propagated along the whole hierarchy.

1.1.3 Benefits

A two-level memory hierarchy is composed of a L1 cache memory (access time: t_{L1}; access energy: E_{L1}) and a L2 memory (access time: t_{L2}; access energy: E_{L2}), with $t_{L1} << t_{L2}$ and $E_{L1} << E_{L2}$.

This memory hierarchy is accessed n_{tot} times and n_{hit} of these accesses are cache hits.

The *hit ratio* is defined as:

$$H := \frac{n_{\text{hit}}}{n_{\text{tot}}} \tag{1.1}$$

The *average access time* and *energy* are defined as:

$$\begin{cases} \bar{t}(H) := Ht_{\text{L1}} + (1-H)t_{\text{L2}} \\ \overline{E}(H) := HE_{\text{L1}} + (1-H)E_{\text{L2}} \end{cases} \tag{1.2}$$

Equation 1.2 shows the criticality of the *hit ratio*: the performance and power consumption advantages provided by the cache are significant if and only if H is sufficiently near to 1.

1.2 Field-Programmable Gate Array

Field-Programmable Gate Arrays are integrated circuits able to implement special purpose circuits described in Hardware Description Language (HDL), thanks to their programmable logic blocks and interconnections.

1.2.1 Memory system

A FPGA memory system is typically made up of:

- Registers: the fastest but most expensive memories, therefore they are only a few.
- BRAMs: on chip Random Access Memories (RAMs) accessible through simple and fast interface.
- External DRAMs: off chip DRAMs accessible through complex and slow interface (e.g. AXI).

1.3 High-Level Synthesis

High-Level Synthesis (HLS) is an Electronic Design Automation technique aimed at translating an algorithm description in a high-level Software programming language (such as C and C++) into a HDL description.

HLS allows designing more complex systems in less time, compared to HDL design, moreover makes the Hardware and Software co-design easier, at the cost of limited low-level control.

This Section is mainly referred to *Vitis™ HLS* [1], but most currently available HLS commercial tools provide equivalent features.

1.3.1 Workflow

The typical HLS workflow consists of:

1. *SW implementation*: the top-level entity is a C function: the function arguments are the entity ports and the functionality is implemented in SW; in order to guarantee synthesizability some constraints should be respected (e.g. no dynamic memory allocation).

2. *SW verification*: the testbench can be developed as a simple main function which calls the top-level entity function, therefore the functionality is verified like any SW: it is possible to exploit traditional tools (e.g. debuggers, print statements. . .).
3. *HW synthesis*: the synthesizer generates a Register-Transfer Level (RTL) description of the top-level entity. It is possible to generate different architectures by setting up some parameters through dedicated directives.
4. *HW verification*: the RTL description is simulated, to make sure that SW and HW outputs match.

1.3.2 Optimization techniques

HLS tools provide different optimization techniques which can be set up by means of compiler directives.

Pipelining

Given a set of sequential stages (e.g. A, B and C of Figure 1.3) which compose an operation (e.g. A + B + C of Figure 1.3) which has to be executed multiple times, the pipelining technique inserts pipeline registers at the output of each stage, so that each stage can run in parallel on different input data (e.g. at the third clock cycle, while C is processing first input, B is processing second input and A is processing third input). The introduced parallelism allows to increase the throughput at a limited additional area cost (only pipeline registers and a FSM are required).

The throughput is determined by the interval (expressed in number of clock cycles) between the beginning of two consecutive executions of the operation, which is called Initiation Interval (II). The optimal pipeline has an II equal to one: at the steady state, one output per clock cycle is produced.

The pipelining can be performed at instruction level, within a loop or a function, or at function level (in HLS terminology this particular kind of pipelining is called *Dataflow*).

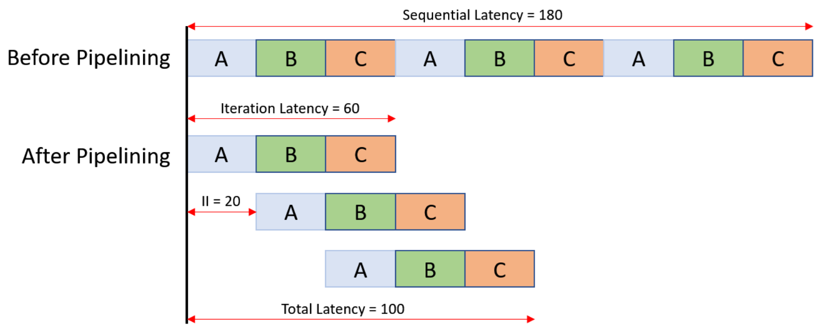


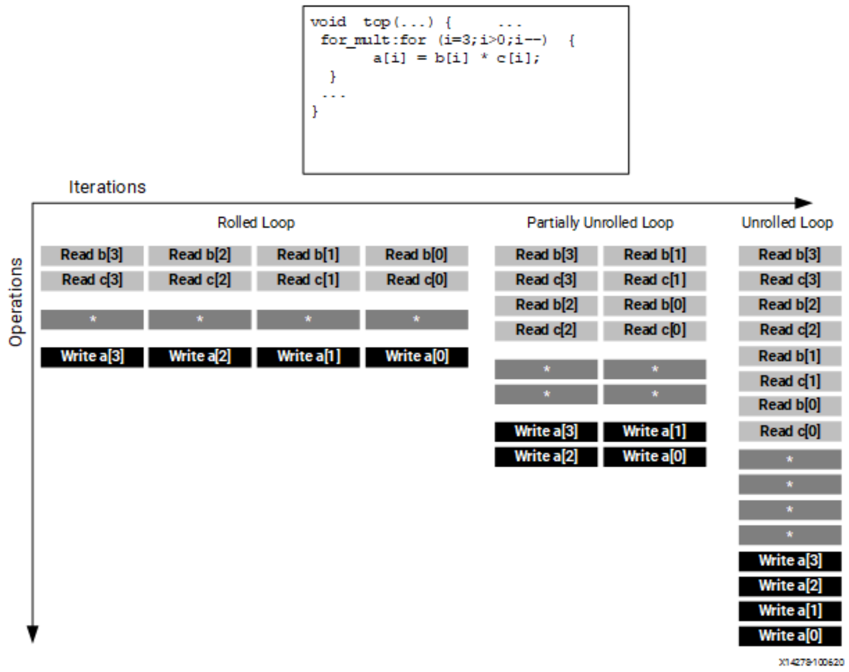
Figure 1.3: Pipelining example.

Loop unrolling

The logic of a rolled loop allows the execution of one iteration at a time: if the loop iterates N times and each iteration has a latency L_{it}, the total loop latency is equal to $L_{loop,rolled} := N \cdot L_{it}$.

The loop unrolling technique instantiates the logic for executing f iterations at a time (where f is the unrolling factor). If there are no dependencies between different iterations, the latency of the unrolled loop is: $L_{loop,unrolled}(f) := \frac{N}{f} \cdot L_{it}$.

Loop unrolling can improve both latency and throughput, but it is expensive in terms of resource usage, since they are multiplied by f.



```
void  top(...) {     ...
 for_mult:for (i=3;i>0;i--)  {
       a[i] = b[i] * c[i];
  }
 ...
}
```

Figure 1.4: Loop unrolling example.

Memory optimizations

- On-chip memory:
  - **Array partitioning**: given a partitioning factor f, an array is split into f portions, each one mapped to a dedicated memory element.

    This allows multiple concurrent accesses to the same array, at the cost of higher memory elements usage.

    Figure 1.5 shows different partitioning modes.
- Off-chip memory:

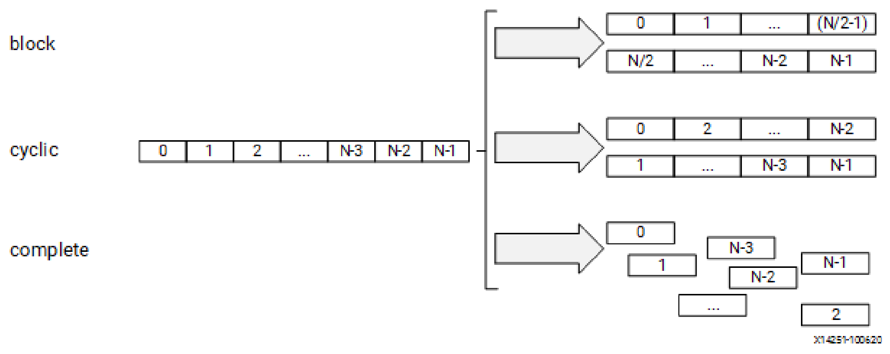


Figure 1.5: Array partitioning examples.

- **Interface widening**: multiple data elements are packed into a single bigger word, to perform multiple accesses at the same time.
- **Burst accesses**: multiple memory accesses are aggregated into AXI bursts to reduce overall latency and improving throughput.

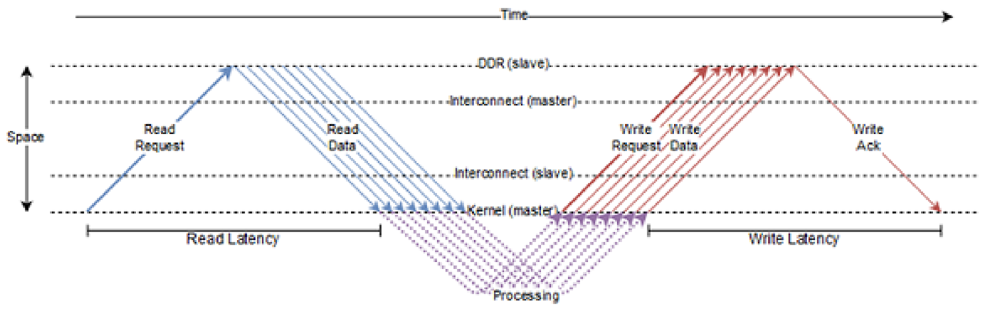


Figure 1.6: Burst read and write example.

2 Development

HLS tools are currently unable to automatically exploit the memory hierarchy present on FPGAs: the only way to take advantage of them is the manual management in a *scratchpad*-like manner, which requires additional design and verification efforts.

The proposed solution **automates the low-level memory management** through a cache module for HLS, which works as an interface with the off-chip DRAM (accessible through an AXI bus) and stores its data to on-chip BRAMs and registers.

2.1 Overview

The proposed cache module has the **dual purpose** of:

- *Reducing the number of DRAM accesses*: misses only needs to access DRAM.
- *Optimizing DRAM accesses*: lines are accessed in bursts through a widened memory interface.

FPGAs provide multiple DRAM ports and HLS can assign each array to a different port: this allows implementing **array-specific** caches, which in general can be easily tuned to reach high hit ratios, since access patterns to a single array are usually regular and there is no interference between accesses to different arrays.

A special attention has been put on **user-friendliness**:

- *Configurability*: cache characteristics can be set through parameters.
- *Integrability*: cache can be inserted into existing designs without requiring many changes.
- *Observability*: critical cache data (e.g. hit ratio) can be profiled during SW simulation for easing the cache parameters tuning.

2.1.1 Ma's cache

Liang Ma et al. proposed a `C++` cache implementation [3] compatible with the *SDAccel™* HLS tool.

It is an array-specific cache module in the form of different `C++` classes: each of them implements an access type (read only/write only and read write) and a mapping policy (direct mapped and set associative).

To improve the *integrability* the `operator[]` has been overloaded so that the cache object can be accessed in the same way as array variables, minimizing the required changes to the code which integrates the cache.

This architecture is **inlined**: the cache logic is directly inserted in the user algorithm logic. This is the major limitation of this solution, since the additional logic inserted in the algorithm may make it too complex and worsen the generated circuit performance.

2.1.2 Proposed solution

The primary goal is to develop the *Basic cache*, a cache architecture which runs in a separate process with respect to the application using it, trying to solve the main limitation of *Ma's cache*: the application logic cluttering due to the inlining.

This architecture has been then optimized in two dimensions:

- *Multi-levels cache*: a L1 cache are added to the cache hierarchy, with the objective of further reducing memory access latency.
- *Multi-ports cache*: multiple cache access points are added to the cache, each one with a dedicated L1 cache, so that multiple requests can be served in parallel.

2.2 Basic cache

The *Basic cache* is aimed at solving the main limitation of *Ma's cache*: application logic cluttering due to inlining.

2.2.1 Architecture

The fundamental idea behind the *Basic cache* is that the cache logic is inserted in a separate process with respect to the application logic accessing it (Figure 2.1): this isolation should make the cache always perform in the same manner, while keeping the application logic as clean as possible, since it would only have to write requests to cache and read its responses, instead of integrating its whole logic.

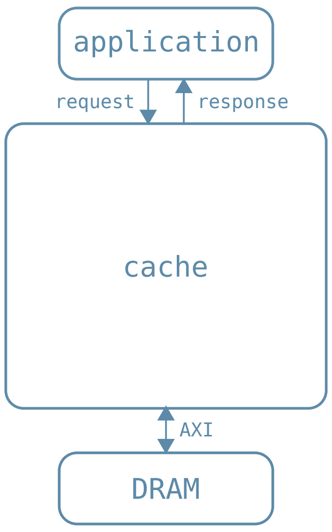


Figure 2.1: *Single-process Basic cache* architecture.

Functionality

If application A needs to access the array associated with the cache C:

1. A sends the access request to C: operation (i.e. read or write), address and (in case of write access) data.
2. C receives the request and checks if the requested address causes a miss.
3. (in case of miss) C prepares its BRAM memory for fulfilling the requested access:
    - (if needed) writes back to DRAM the BRAM line to be replaced.
    - reads from DRAM the requested line and store it to BRAM.

4. C performs the requested access to BRAM and (in case of read request) sends requested data to A.

Characteristics

The *Basic cache* is compliant with the set associative mapping policy and the write-back consistency policy. It is configurable in terms of:

- Word type and number of words per line.
- Number of sets and ways (therefore, it is possible to obtain a fully associative policy by setting the number of sets to 1 or a direct mapped policy by setting the number of ways to 1).
- Replacement policy (Least Recently Used or First-In First-Out).

Single-process Basic cache

The *Single-process Basic cache* is composed of a single pipelined process which performs all the cache functionalities.

This process can be pipelined with an II equal to 1 when memory accesses are Read-Only and a cache line can fit a single AXI transaction (i.e. line is not bigger than the maximum AXI interface width: 512 or 1024 bits typically, depending on the specific device).

Write accesses generate some dependencies on the AXI interface, while large cache lines require multiple AXI transactions: both of them cause an increase of the process II, ruining cache performance.

Multi-processes Basic cache

The *Multi-processes Basic cache* splits cache into two processes (Figure 2.2):

- *Core* process: manages communication with application and keeps cache data structures up to date.
- *Memory interface* process: deals with the AXI interface.

This architecture is aimed at solving the performance limitations of the *Single-process Basic cache*: it manages to pipeline the *core* process with an II equal to 1, even in case of write accesses or long lines, since the AXI interfacing resides in the separate *memory interface* process.

The latency of the response to a hitting request depends on the *core* process only, therefore with this solution the best performance is achieved in case of writable caches too.

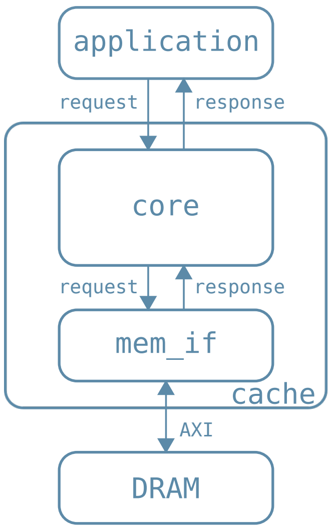


Figure 2.2: *Multi-processes Basic cache* architecture.

2.2.2 Implementation

The *Basic cache* is implemented in the form of a `C++14` [2] class, compatible with *Vitis™ HLS 2021.1*. All the configurable parameters are set through class template arguments.

The cache class is logically split into two parts:

- *Internals*: cache functionalities.
- *Interface*: APIs for managing requests and responses from application side.

Internals

The *Internals* implementation differs between the *Single-process* and the *Multi-processes* implementations:

- *Single-process Basic cache*: single process which implements all the cache functionalities.
- *Multi-processes Basic cache*:
  - *Core* process: same as *Single-process Basic cache* process, but it does not directly access the AXI bus: it issues requests to the *memory interface* process through FIFOs.
  - *Memory interface* process: it accesses the AXI bus as requested by the *core* process.

Single-process Basic cache, with respect to the *Multi-processes* one, requires lower resource usage and better performance, when it is possible to schedule its process with an II equal to 1 (read-only accesses with line not larger than the maximum AXI interface bitwidth): therefore it is automatically instantiated whenever it is convenient.

Process modeling HLS is intended for synthesizing sequential Software code, therefore it has been necessary to develop a novel technique for modeling multiprocess designs.

A process is modeled as an infinite loop and the parallelism between multiple processes is modeled differently depending on the compilation target:

- *SW simulation*: each process is mapped to a `std::thread`.
- *HW synthesis*: each process is a dataflow function, in a dataflow region with the `disable_start_propagation` option disabled (which allow each function to run in parallel, without waiting for the completion of previous ones).

The distinction between simulation and synthesis code can be performed through the "`#ifdef __SYNTHESIS__`" preprocessor directive.

Different processes can communicate by means of FIFOs (`hls::stream` provided by Vitis™HLS), which allow unidirectional point-to-point communication between two processes. It is possible to insert multiple FIFOs between each process, in both directions, therefore allowing to set up duplex communication.

Since `hls::stream` provides blocking operations, these FIFOs can be also used for synchronization purposes.

Dataflow checking Alternatively executing the *Multi-processes* or the *Single-process* code with traditional `if` statements would generate errors during the synthesis, particularly in the *Dataflow check* step (which checks if each `hls::stream` has a single reader and a single writer): the compiler builds both branches of the `if` statements, independently of the fact that one of them is never executed.

The problem has been solved through a wrapper class, which conditionally includes a `hls::stream` object, exploiting the template specialization mechanism.

Arrays partitioning Cache memory (which stores the actual data) must be accessed one line per clock cycle: it is mapped to a BRAM array cyclically partitioned with a factor equal to the number of lines.

Helper data (e.g. `tag`, `valid`, `dirty`...) is stored to completely partitioned arrays, mapped to registers, in order to avoid dependencies as much as possible and get the best performance.

Read After Write dependencies The cache process II is limited by the RAW dependencies on the cache lines, therefore the *RAW cache* has been developed: it is a single-line cache which provides the functions:

- `get_line`: in case of hit, read the *RAW cache* line; in case of miss, read the cache line.
- `set_line`: write both the *RAW cache* line and the cache line.

Cache memory is always accessed through the *RAW cache* and the `set_line` function is called once per iteration at most: if a cache line has been written, it is impossible that it is read in the next iteration, since the RAW cache would hit and return its line. This allows to falsify the RAW dependency of distance of 1 on the cache memory, making it possible to schedule the cache process with an II equal to 1.

Interface

Interface provides APIs for managing requests and responses from application to cache:

- `get`: send a read request and read the response.
- `set`: send a write request.

To improve user-friendliness, similarly to *Ma's cache*, the `operator[]` has been overloaded so that a cache object can be used as a traditional array (e.g. `val = cache[i]` calls `val = cache.get(i)` and `cache[i] = val` calls `cache.set(i, val)`).

2.3 Multi-levels cache

The *Multi-levels cache* purpose is to improve the memory hierarchy performance by adding a faster L1 cache memory on top of it.

2.3.1 Architecture

The Inter-Process Communication (IPC) (i.e. FIFO accesses) introduces an overhead at each memory access, therefore the L1 cache is inlined in the application logic (Figure 2.3), so that there is no need of any IPC mechanism.

In order not to fall into the same cluttering issues of *Ma's cache*, the L1 cache is kept as simple as possible:

- Mapping policy: direct-mapped.
- Consistency policy: write-through.

The write-through consistency policy discards any advantage for write accesses, but given that simplicity is a priority and read accesses are usually much more frequent than write accesses, this has been considered the best trade-off.

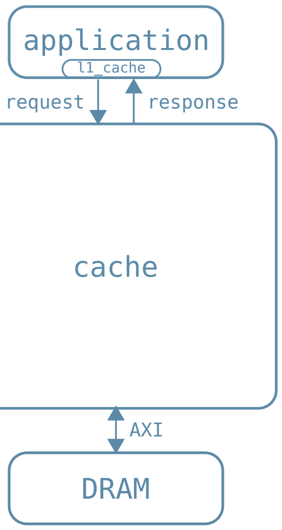


Figure 2.3: *Multi-levels cache* architecture.

2.3.2 Implementation

The *Multi-level cache* has been implemented adding the L1 cache to the *Basic cache*. It is possible to configure the number of L1 cache lines through the `L1_CACHE_LINES` template parameter, therefore by setting it to 0 the resulting architecture is equivalent to the *Basic cache*.

Internals

The only difference with respect to the *Basic cache* implementation is that the response to a read request does not send a single word, but a whole cache line (therefore the data FIFO flowing from cache to application has been widened accordingly).

Interface

The L1 cache is contained in the *Interface*: the newly introduced `get_line` function receives an address A in input and it returns the line to which A belongs. In particular, it first checks if A hits in the L1 cache: if so it reads the data from the L1 cache, otherwise it issues the request to the L2 cache.

It is still possible to use the same *Basic cache* APIs, which have been updated to support the L1 cache:

- `get`: it calls the `get_line` function and then returns the requested word.
- `set`: it sets L1 cache line to dirty, if it hits, and it issues the request to the L2 cache.

2.4 Multi-ports cache

The computational core of many algorithms consists in a loop, which HLS can optimize with two techniques: *Pipelining* and *Unrolling*.

The *Basic* and *Multi-levels* caches are suitable for *Pipelining* since they complete one access per clock cycle, at the steady state, in case of hit, however they are not suitable for *Unrolling*, since they do not support concurrent accesses.

The *Multi-ports cache* has been specifically designed for adding support to multiple concurrent accesses to the same cache memory, allowing to efficiently unroll application loops.

2.4.1 Architecture

The *Multi-ports cache* is characterized by multiple access ports (Figure 2.4).

Each port has dedicated logic for accessing the L2 cache and an independent L1 cache. L1 caches comply with the write-through consistency policy, therefore to guarantee coherency between the multiple caches it is enough to inform all the L1 caches about write accesses, so that they invalidate their line in case of hit.

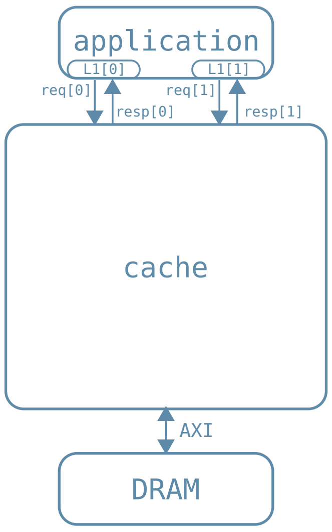


Figure 2.4: *Multi-ports cache* architecture.

2.4.2 Implementation

The *Multi-ports cache* has been implemented extending the *Multi-levels cache*.

It is possible to configure the number of ports through the `PORTS` template parameter, therefore by setting it to 1 the resulting architecture is equivalent to the *Multi-levels cache*.

Internals

To avoid dependencies issues, whenever `PORTS` is greater than 1, the *Multi-process* internals architecture is generated.

The *Core* process has been modified to serve requests coming from all the ports by inserting an unrolled loop which iterates over all the ports. HLS guaranteees all the dependencies on cache data structures, and the resulting II of the *Core* process is equal to `PORTS`.

Interface

FIFOs between *Core* and application and L1 cache have been replaced with arrays of FIFOs and L1 caches, completely partitioned, so that they are independent.

Each call to `get_line` (which is in turn called by `get`) and `set` is associated with a specific port by means of a member variable holding the port index and is updated after each access.

2.4.3 Limitations

In some particular situations (e.g. cache is explicitly accessed multiple times per iteration) the co-simulation enters a deadlock. The source of this problem can be probably found in the port indexing and to be fixed may require more control over the operations scheduling, which is not provided by *Vitis™ HLS 2021.1*.

3 Results

3.1 Matrix multiplication

3.2 Bitonic sorting

3.3 Lucas-Kanade

4 Conclusion

Bibliography

[1] Xilinx Inc. *Vitis High-Level Synthesis User Guide*. Aug. 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug1399-vitis-hls.pdf.

[2] ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. Fourth. Dec. 2014.

[3] Liang Ma, Luciano Lavagno, Mihai Lazarescu, and Arslan Arif. “Acceleration by Inline Cache for Memory-Intensive Algorithms on FPGA via High-Level Synthesis”. In: *IEEE Access* PP (Sept. 2017), pp. 1–1. DOI: 10.1109/ACCESS.2017.2750923.